UKRAINIAN CATHOLIC UNIVERSITY

BACHELOR THESIS

Fragments of formal verification of the Solidity smart contracts

Author:
Maksym SHUMAKOV

Supervisor:
Vasyl LENKO

A thesis submitted in fulfillment of the requirements for the degree of Bachelor of Science

in the

Department of Computer Sciences
Faculty of Applied Sciences

Lviv 2022

Declaration of Authorship

I, Maksym SHUMAKOV, declare that this thesis titled, "Fragments of formal verification of the Solidity smart contracts" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

"The lessons of all the failures, and the process of examining them and their implications, were the lifeblood of mathematics. In that sense, failure is nothing more than the history of the proof of the hypothesis. As Mao puts it, the logic of the people is 'fight, fail, fail again, fight again... till their victory'. "

Alain Badiou

UKRAINIAN CATHOLIC UNIVERSITY

Faculty of Applied Sciences

Bachelor of Science

Fragments of formal verification of the Solidity smart contracts

by Maksym SHUMAKOV

Abstract

We present our elaborations on the software verification techniques using the Coq formal proof management system of the Solidity smart contracts. As modern Blockchain systems that provide smart contracts functionality often manage enormous amounts of valuable digital assets, it is significant to understand the inherent vulnerabilities of the system's general architecture and implementation. We have confined our model only to Solidity and analogues depth-first execution blockchains. Our research considers the Safe Remote Purchase Solidity contract and modelled in Coq's functional language Gallina.

Acknowledgements

I express gratitude to my supervisor Vasyl LENKO, my teacher Oleh FARENYUK, and all the stuff of the Ukrainian Catholic University for instilling in me the enthusiasm to explore complex and poorly understood areas of science.

Contents

List of Figures

List of Tables

List of Abbreviations

DyCS	**D**istributed yet **C**entralized **S**ystem
P2P	**P**eer-to-**P**eer
EVM	**E**thereum **V**irtual **M**achine
FV	**F**ormal **V**erification
FPMS	**F**ormal **P**roof **M**anagement **S**ystem
SRPC	**S**afe **R**emote **P**urchase **C**ontract

Dedicated to the heroic resistance of the Ukrainian people to Russian fascism

Chapter 1

Introduction

1.1 Overview

As modern Blockchain systems that provide smart contracts functionality often manage enormous amounts of valuable digital assets, it is significant to understand the inherent vulnerabilities of the system's general architecture and implementation. There are many methods to verify the correctness of the program, but usually all of them are reduced to testing on input data, which is very inefficient and does not give a broad understanding of how the system works and what its shortcomings. This applies to blockchain systems.

Formal verification is gaining momentum in use in complex systems. In recent years, many attempts have been made to formalize such systems as blockchain and smart contracts. this usually requires significant simplifications and abstractions from how the system actually works.

However, it is necessary to develop such a thorough and fundamental approach to the areas on which finances, people's lives or the democracy of society depend. In our opinion, formal verification is a fundamental method that has deep potential for testing such systems.

1.2 Proposed Solution

We use the mathematical proof system Coq, which uses the functional Gallina programming language, to model some aspects of the formal verification of smart contracts. We have confined our model only to Solidity and analogues depth-first execution blockchains. Our research considers the Safe Remote Purchase Solidity contract and modelled in Coq's functional language Gallina.

Chapter 2

Blockchain and Solidity Smart Contracts

2.1 General overview

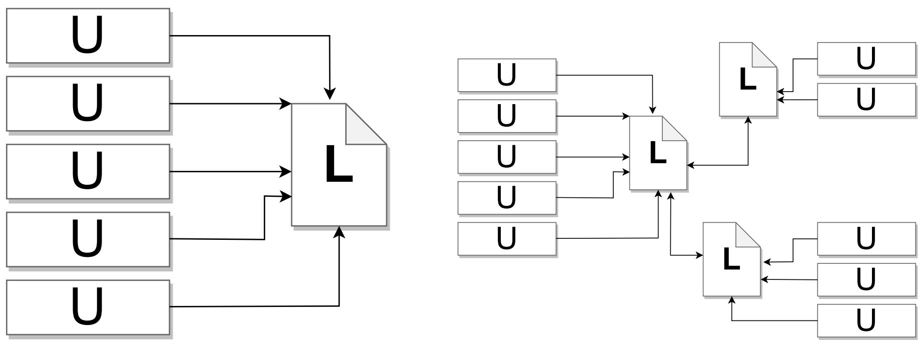
In accounting systems, business transactions are stored in accounting journals and ledgers. When the journal is filled with information in the system, it is moved to the ledger. As it is critical for companies, organizations and businesses to maintain the proper state of their accounts, ensuring the reliability and accuracy of the records in the van is a priority. Accounting is a model that can design systems that are not directly related to finance—for example, ensuring the security of rights for referendums, polls, and consensus decision-making. Thus, the issue of ledgers becomes, for instance, a matter of ensuring the democracy of society.

In addition to transaction records, accounting systems can reproduce stored information. This allows owners to track the transfer of assets, the availability of an asset to the user, forecasting and more. Historically, there are two types of such systems based on the type of their ledgers [19]:

- **single-entry**: a single record carries out the maintenance of accounting requirements in the accounting journal. We meet such systems every day when we receive confirmation of the purchase of goods in the form of a check, etc. This approach is convenient for small businesses, community organizations and similar organizations. However, single entries have drawbacks, such as the lack of a method to record the status of user accounts. Because of this, it is difficult to assess the system's general state, which in some cases can lead to fraud and such use of the system, which the designers do not provide.

- **double-entry**: all accounts contain a table of two columns: debit and credit. When completing a transaction, the user who redirects the asset rules out it from his account (credits), and the receiver adds to his (debits). As a result, we have a display of each operation in two records. It significantly expands the possibilities of accounting, as it becomes possible to track suspicious activity and the system's overall status. Another essential characteristic is the invariant of the assets:
$$\text{Assets} = \text{Liabilities} + \text{Equities}$$
At the end of a set of transactions in the system, the total debit and credit must be equal. Any deviation from this equation will highlight an error.

Ledger centralization is considered based on two topology characteristics: centralization of *control* and *location*. Both system control and localization can be centralized or decentralized. Combining these aspects of the system determines its vulnerabilities, advantages and connectivity.



(A) centralized

(B) decentralized

FIGURE 2.1: Two versions of topology of ledger control. **U** stands for user, **L** for ledger.

Fig.2.1 shows the topology of a centralized and decentralized approach to the organization of control over ledgers. In approach (A), we have a system in which one of the graph's vertices representing the structure has complete control over the ledger and therefore establishes control over all network participants. In turn, in approach (B), we have a decentralized practice in which several mutually independent entities exercise control. In this case, there is no single medium of all information, which has both advantages and disadvantages.

Another essential characteristic of implementing such systems is the physical location of its elements. We will call this *centralization by location*. *Centralized by location* is a system in which all its parts (leader, servers, devices, etc.) are located in one geographical region. *Distributed* is one in which its functional parts do not have a single location.

The relationship between these types of organizations of accounting systems determines their types. Table 2.1 shows which type of organization is characterized by which place and type of control. Two of them are important for our topic: *DyCS* and *distributed*.

- **DyCS**: the type of organization in which the arrangement of equipment and means of storing information about transactions is distributed by location but controlled by a single user or another entity. This approach is often used to implement cloud access systems, where the provider has control over the entire network.

- **Distributed**: in this approach, both location and control of the ledger are distributed. There is no single top of the network that has complete control over assets or other resources. Implementing such an architecture is a P2P network in which all users are independent of each other, and there is no functional centre.

Blockchain is a P2P network with a distributed double-entry ledger, storing information about all transactions and events. Anyone on the network can access Ledger, but read-only. Transactions that have already taken place are immutable and cannot be changed [10]. Changes to the blockchain are made through a consensus system, which does not presuppose centralized control. It is replaced by collective management provided through the protocols inherent in the system. After making a consensus decision, information about the blockchain is transmitted over the P2P network to all its users. Implementing a blockchain requires a considerable amount of functionality, such as smart contracts and sophisticated cryptography.

	Centralized by location	**Distributed by location**
Centralized control	Centralized system	DyCS
Decentralized control	-	Distributed system

TABLE 2.1: Relations between control and location centralization types

2.2 Layered Blockchain Architecture

There are many blockchain implementations (Bitcoin, Ethereum, Litecoin, etc.). All of them more or less fit into the layered structure of the blockchain, which is an abstraction that allows one to divide the functional parts of the blockchain system into several levels (layers) depending on their complexity and features. Fig.2.2 shows the classification of the layers of the blockchain, and the arrows indicate that the current layer is inferior to the successive in its abstraction level [19].

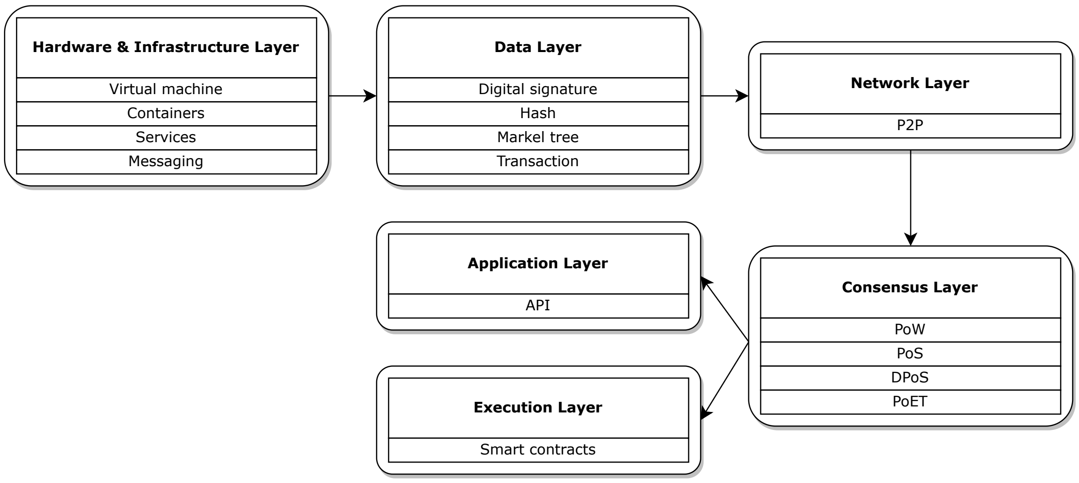


FIGURE 2.2: Blockchain layered architecture with separated application and execution levels

- **Hardware and Infrastructure Layer**: is responsible for the integration of individual computers into the P2P blockchain network. Each such client is represented as a node responsible for the validation of transactions and the organization of blocks. Each node has a current copy of the ledger, updated as changes are made by consensus to the blockchain. Each node is decentralized and distributed. Take, for example, Ethereum. To initialize a new node in the network, one needs to download the client to the machine and run the EVM to perform actions as a client. In addition, the EVM allows the deployment of Ethereum smart contracts.
- **Data Layer**: the data layer is responsible for implementing the blockchain data structure. It presents itself as a linked list and pointer. The linked list arranges the transaction blocks (Fig.2.3), containing a pointer to the previous block in the chain, the Merkle root of the tree, and a list of transactions. The Merkel

tree stored in the block contains its hash and additional cryptographic information. Hashing differs depending on the implementation. The still usually used algorithm is The SHA-256.

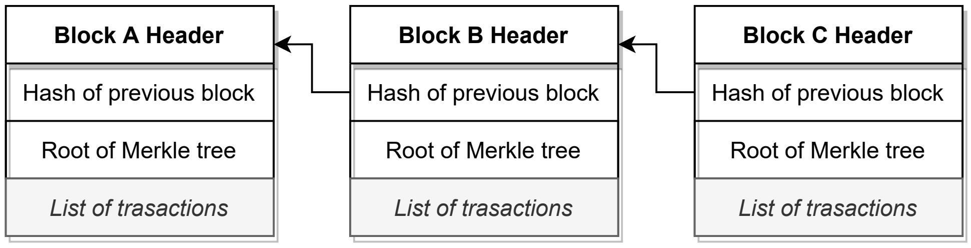


FIGURE 2.3: Blockchain data structure example

- **Network Layer**: the network layer is responsible for network communication between nodes. It contains functionality for finding, communicating, synchronizing nodes with each other in the P2P network.

- **Consensus Layer**: the level of consensus is one of the most critical components of any blockchain. At this level, the part of the system responsible for the adoption of consensus, and hence proper functioning, operates between the nodes of the P2P network of a particular blockchain. This level is responsible for synchronizing the nodes with the ledger, monitoring the distribution of the system to prevent "capturing" it, recognizing the truth of a decision and many other functions. The consensus layer is at the heart of the blockchain, so different providers offer different approaches. For example, Ethereum implements a system with probabilistic consensus, which allows a situation in which not all users will have a current copy of the ledger, which leads to a split of the blockchain. Some other providers use a deterministic version of the algorithm [21].

- **Application and Execution Layers**: this layers of blockchain are often inseparable, but with the introduction of Ethereum smart contracts, where these levels are two distinct, it is considered good practice to see them as two different. These two layers contain the entire functional part through which the user interacts with the blockchain. The application layer provides API, allowing one to work with the system efficiently. The execution layer is wholly dedicated to the execution of smart contracts.

2.3 Solidity Smart Contracts

All nodes in the P2P blockchain network are divided into two types: users and smart contracts. Each node has its unique address by which it can be identified. Some blockchain implementations distinguish between addresses of these two types (such as Ethereum), and some do not. So a smart contract is a node in a decentralized network, but it is not managed from the outside by any agent. Instead, it contains program code that, under certain circumstances, runs and can change its state.

When creating a transaction node, one must specify the address of the recipient, as well as, if necessary, a message in which to provide additional information about this transaction. It will be implemented or not by consensus. However, if the address specified in the contract does not exist, a new smart contract will be created with the code encrypted in the transaction message. Also, in such messages, the input data for the execution of the program code of the smart contract are transferred [17].

Each smart contract has its state, a local configuration of information in variables, fields and storage that it stores. When one calls the execution of the smart contract code, its status may change. The smart contract's state is an essential tool for representing the status of real objects, such as the distribution of wages in the company, Financial derivatives, and the contents of files in the archives. Transactions are the only way to trigger code execution in a contract, and any state changes that will result will be stored in the blockchain [19][12].

Smart contracts can be designed in different languages, but the language proposed by Ethereum, Solidity, was made specifically for this purpose. The user-written source code is compiled using the Ethereum runtime engine. This allows one to use the created bytecode for efficient execution on EVM. Solidity, in many respects, follows the syntax of JavaScript and provides extensive functionality for the implementation of smart contract programs.

```
contract SimpleStorage {
    uint storedData;

    function set(uint x) public {
        storedData = x;
    }

    function get() public view returns (uint) {
        return storedData;
    }
}
```

FIGURE 2.4: Solidity `SimpleStorage` smart contract example

In Fig.2.4 [17] we have a smart contract `SimpleStorage`, which implements storage for a single number of type `uint`. The variable `uint storedData` determines the state of the contract. Every participant of the network has an access to get the state of the contract by calling the `get()` function marked with the `view` keyword indicating that the function does not change the state. Via `set(uint x)` function one can modify the state by assigning another value `x` of the type `uint` to the variable `uint storedData`. This can only be done by creating the transaction with proper input data encoded in a transaction message.

Chapter 3

Formal Verification

When designing software and hardware technology, the development team may make mistakes or not anticipate other vulnerabilities in their technology. Most modern software contains vulnerabilities. Moreover, due to the principle of backward compatibility, each such error is signalled in the future, requiring much more resources to circumvent it. Checking the correctness of the program requires testing it on all sorts of input data. A similar approach is used in unit testing. Moreover, even though the possible combinations of input data are finite, their number usually makes it impossible to test the correctness of the technology effectively. Dijkstra writes in his book [4]:

> As I have now said many times and written in many places: program testing can be quite effective for showing the presence of bugs, but is hopelessly inadequate for showing their absence.

Another approach that differs from checking the correctness of all input data is formal verification. FV is an approach in which an algorithm, program or hardware is modelled using mathematical entities. Having a collection of such objects, it becomes possible to prove the properties of the system using the methods of mathematical logic. It is evident that the characteristics proved in this way will be valid regardless of the input data type.

Despite the theoretical sophistication of FV and the ability to prove general statements about the technology, it has a number of shortcomings that prevent its widespread use in applied fields [6]:

- Designing applications into a formal system is often a task in which the initial structure of the project is lost to ensure the proper level of abstraction and generality. Because of this, the mathematical model is confusing, complex and problematic for financing.
- Proving claims about the properties of the application system is a difficult task. The more complex the formalization, and the formal systems that fit well the real ones are often very complex, the more sophisticated the skills required in a mathematical proof.
- The correctness of the formal system does not necessarily entail the correctness of the real one. During formalization, some aspects may be omitted, which significantly affects the final result of verification.

Despite significant shortcomings, formal verification remains the most reliable method of proving the correct operation of programs and hardware. It is widely used in developing processors, aircraft technology and more. FV requires some system that will act as a mathematical model. Consider an approach that uses typed lambda calculus and intuitionistic logic for modelling.

3.1 Propositions As Types

3.1.1 Natural Deduction

Natural deduction is a formalization of the derivation system that mathematicians use in practice. Gentzen proposed the natural deduction in the 1930s due to its dissatisfaction with the then widely used Hilbert system. The latter proposed a set of axioms and inference rules based on which deduction could be formalized. However, the process of working within the Hilbert system did not resemble the real work of a mathematician. It was instead a rigid system of formalisms [13].

In turn, Gentzen set himself to get as close as possible to the usual, intuitive understanding of deduction. The system of natural deduction he proposed is the Gentzen `NJ` system. It does not contain axioms but only inference rules. The latter defines the content of logical connections.

A language of *terms* is build using the following grammar [5]:

$$\texttt{Terms}\ \ t ::= x \mid a \mid f(t_1, t_2, \ldots, t_n)$$

where x is a *variable*, a - *parameter* and f is a *functional symbol* of arity n. Therefore the possible terms are: $f(x, g(a, b))$, $f(g(f(x, x, a, h(c))))$, etc.

The language of *propositions* is built up from predicate symbols P, Q, etc. and terms in the usual way:

$$\begin{aligned}\texttt{Propositions}\ \ A ::= & P(t_1, \ldots, t_n) \mid A_1 \wedge A_2 \mid A_1 \rightarrow A_2 \mid A_1 \vee A_2 \mid \\ & \neg A \mid \top \mid \bot \mid \forall x.A \mid \exists x.A\end{aligned}$$

In natural deduction each logical connective and quantifier is uniquely determined by two rules: *introduction* and *elimination* rules. Moreover, it cannot use other connectives and quantifiers in its definition. This principle is called *orthogonality* stipulating that basic blocks of natural deduction are all independent of each other.

- **Introduction rule**: determines how to infer the truth of logical connective or quantifier (i.e. conjunction, implication) based on the truth-value of the inputs.
- **Elimination rule**: deduces the truth-values of other propositions given that the connective is true.

A connective is considered to be defined if it is provided with both rules (introduction and elimination) such that they satisfy two properties:

- **Local soundness**: when introducing a connective and then immediately eliminating it, we should be able to erase this detour and find a more direct derivation of the conclusion without using the connective.
- **Local completeness**: we can eliminate a connective in a way which retains sufficient information to reconstitute it by an introduction rule.

Both these properties require introduction and elimination rules to be "equally strong" for we could restore the initial truths or the connective after eliminating it.

For clarity, define a logical conjunction. To do this, one must specify the rule of introduction and the rule of elimination. From our intuition of logic we define conjunction introduction rule as: the conjunction is true when both of the operands are true. Thus the following introduction rule [23]:

$$\frac{A\ \texttt{true} \qquad B\ \texttt{true}}{A \wedge B\ \texttt{true}} \wedge\mathrm{I}$$

This leads to two elimination rule, namely $\wedge E_L$ for left elimination and $\wedge E_R$ for right:

$$\frac{A \wedge B\ \texttt{true}}{A\ \texttt{true}}\ \wedge E_L$$

$$\frac{A \wedge B\ \texttt{true}}{B\ \texttt{true}}\ \wedge E_R$$

Using these rules we can attest if local soundness and local completeness are satisfied. If not these rules for conjunction cannot be accepted in natural deduction system.

$$\dfrac{\dfrac{\overset{\mathcal{C}}{A\ \texttt{true}} \qquad \overset{\mathcal{D}}{B\ \texttt{true}}}{A \wedge B\ \texttt{true}}\ \wedge \mathrm{I}}{A\ \texttt{true}}\ \wedge E_L \quad \Longrightarrow_R \quad \overset{\mathcal{C}}{A\ \texttt{true}}$$

Analogously the inference of B in the context $\mathcal{D}$ can be achieved. Clearly, there is no need to proceed with all these deductions to infer A `true`, thus the local soundness holds.

As for local completeness we can use this deduction tree:

$$\overset{\mathcal{E}}{A \wedge B\ \texttt{true}} \quad \Longrightarrow_R \quad \dfrac{\dfrac{\overset{\mathcal{E}}{A \wedge B\ \texttt{true}}}{A\ \texttt{true}}\ \wedge E_L \qquad \dfrac{\overset{\mathcal{E}}{A \wedge B\ \texttt{true}}}{B\ \texttt{true}}\ \wedge E_R}{A \wedge B\ \texttt{true}}\ \wedge I$$

As a result, the rules of introduction and elimination that we have defined for the conjunction are suitable for use in natural deduction. So we have the definition of logical and. This simple mechanism allows one to determine all the logical connections and quantifiers to work with intuitive logic. The transition to classical logic requires the introduction of the rule of excluding the third, but there are some subtleties that we will not cover in this paper [13]. We present in the Table 3.1 the rules of introduction and elimination to determine other elements of Gentzen's natural deduction.

3.1.2 Lambda Calculus

In the early 20th century, Whitehead and Russell, in their remarkable *"Principia Mathematica"* [22], showed that the methods of logic could serve to represent much of the mathematical objects and judgments. This became the basis for many areas of research. For example, in philosophy, this gave new impetus to logical positivism, the philosophical school of epistemology, which relies on the methodology of mathematical and logical apparatus for the study of real-world phenomena [9].

The discoveries of Whitehead and Russell had a significant impact on the German mathematician David Hilbert and the scientific school he founded. In mathematical discourse, Gilbert posed a new problem, *Entscheidungsproblem* (decision problem), which consisted of proposing an *"efficiently calculable"* method for determining the truth value of any logical proposition. The central mystery of the *Entscheidungsproblem* was determining what is considered *"efficient calculable"* [20].

One of the attempts to define "efficiently calculable" is *lambda calculus* invented by Alonzo Church at Princeton. The initial aim of the lambda calculus is to provide calculable system to encode and work with logical propositions. However, its applications go beyond the scope of the original goal.

We can define the syntax of the lambda calculus using the Backus–Naur form [16]:

$$
\begin{aligned}
\texttt{<expression>} &::= \texttt{<name>} \mid \texttt{<function>} \mid \texttt{<application>} \\
\texttt{<function>} &::= \lambda\texttt{<name>}.\texttt{<expression>} \\
\texttt{<application>} &::= \texttt{<expression>}\texttt{<expression>}
\end{aligned}
$$

FIGURE 3.1: Lambda calculus syntax grammar

where `<name>` is an element from the set $\mathcal{V}$ called *names* or more usually *variables*.

Functions are evaluated when there is an expression in which the first argument is a function and the second is arbitrary expression. The result of evaluation is a substitution of the expression in the place of the function's name in functional expression. For instance, consider the expression produced by the grammar (Fig.3.1):

$$(\lambda x.(\lambda t.xtx))E_1E_2$$

Using substitution for evaluation of the lambda function we, firstly, apply function $(\lambda x.\dots)$ to E_1:

$$(\lambda x.(\lambda t.xtx))E_1E_2 \equiv (\lambda t.E_1tE_1)E_2$$

and then continue this process with another function acting on the expression E_2:

$$(\lambda x.(\lambda t.xtx))E_1E_2 \equiv (\lambda t.E_1tE_1)E_2 \equiv E_1E_2E_1$$

Another interesting application of lambda functions, besides logical proposition encoding, is arithmetic formalization [16]. There is a way to represent non-negative integers using the lambdas. Let us write $\lambda x.(\lambda y.E)$ as $\lambda xy.E$. In the sense of computation these two notions are absolutely equivalent. Therefore, we can define 0 as:

$$0 := \lambda sz.z$$

and all the successive numbers are defined as a number of applications of s to z:

$$N := \lambda sz.\underbrace{s(s(\dots s(z)\dots))}_{N \text{ times}}$$

Now we can define lambda function $\texttt{S}$ representing the successive number to the given:

$$\texttt{S} := \lambda wyx.y(wyx)$$

When one tries to compute $\texttt{S}0$ he or she gets 1 as expected:

$$
\begin{aligned}
\texttt{S}0 &\equiv (\lambda wyx.y(wyx))(\lambda sz.z) \\
&\equiv \lambda yx.y((\lambda sz.z)yx) \\
&\equiv \lambda yx.y((\lambda z.z)x) \\
&\equiv \lambda yx.y(x) \equiv 1
\end{aligned}
$$

One can you induction to show that for any N defined using the lambda functions the following expression is true:

$$\mathtt{S}N \equiv N + 1$$

3.1.3 Type Theory

Type theory arises from the paradox of set theory, which was considered a candidate for the position of the fundamental theory of mathematics discovered by Bertrand Russell. The paradox arises in a situation where the predicate can be applied to itself, such as in the statement "a set that contains all sets but not itself." Written in the set-theoretic style:

$$R = \{x \mid x \notin x\}$$

If we suppose that $R \in R$ then by the definition of R it should not be an element of itself. Still, on the other hand, if $R \notin R$ then it must belong to itself, namely $R \notin R \implies R \in R$. To avoid this paradox Russel proposed a *type theory*.

If the set theory has a dual structure, namely sets and language, that allows them to operate, then type theory operates only with types. Another difference is that the set is uniquely characterized by the elements that inhabit it, while the type does not have such a property. In type theory, each element has its type, outside of which it does not exist [7]. By convention, this is written as $a : A$ meaning that element a has a type A.

Types as such also have their types. Types of types inhabit the universe, which is strictly ordered.

$$\mathcal{U}_0 : \mathcal{U}_1 : \mathcal{U}_2 : \ldots$$

The structure of types is cumulative, i.e. the object that is of a type $\mathcal{U}_i$ is also of a type $\mathcal{U}_{i+n}$ for arbitrary non-negative integers i, n.

This hierarchy is precisely the means to avoid Russell's set paradox, because all mathematical objects are assigned a type and being on the one level of this order object cannot operate on the higher.

3.1.4 Curry-Howard Isomorphism

One of the most important results of mathematical logic for formal verification is the *Curry-Howard correspondence*, which finds a fundamental connection between logical statements, types, and lambda calculus. Usually this isomorphism is described by the following statements:

- **Propositions as types**: meaning that for every logical proposition there is a type that corresponds to it.
- **Proofs as programs**: states that to provide a proof for a certain logical proposition it is enough to provide a program of that type. We work with lambda calculus to write programs, hence every lambda function as its type.
- **Simplification of proofs as evaluation of programs**: to evaluate a program in some sense means to simplify a logical proof and vice versa.

As an example of application of Curry-Howard isomorphism we will provide a proof of the following proposition of intuitionistic logic $A \wedge (A \wedge A \rightarrow B) \rightarrow B$.

Using the means of natural deduction and the corresponding introduction and elimination rules for the connectives and quantifiers we can prove a proof [15]:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \wedge (A \wedge A \to B)}\,u}{A \wedge A \to B}\,\wedge E_R \qquad \dfrac{\dfrac{}{A}\,w \quad \dfrac{}{A}\,w}{A \wedge A}\,\wedge I}{B}\,\to E}{A \to B}\,\to I^w \qquad \dfrac{\dfrac{}{A \wedge (A \wedge A \to B)}\,u}{A}\,\wedge E_L}{\dfrac{B}{A \wedge (A \wedge A \to B) \to B}\,\to I^u}\,\to E$$

Now, having the proof of the proposition we can use deduction rules (Fig.3.2) described in [15] to get the corresponding program in lambda calculus. By the Curry-Howard correspondence that program will be equivalent to the derivation above.

$$\dfrac{M : A \quad N : B}{\langle M, N\rangle : A \wedge B}\,\wedge I \qquad \dfrac{M : A \wedge B}{\textbf{fst}\ M : A}\,\wedge E_1 \qquad \dfrac{M : A \wedge B}{\textbf{snd}\ M : B}\,\wedge E_2$$

$$\dfrac{\begin{array}{c}\overline{u : A}\\ \vdots \\ M : B\end{array}}{\lambda u : A.\ M : A \supset B}\,\supset I^u \qquad \dfrac{M : A \supset B \quad N : A}{M N : B}\,\supset E$$

$$\dfrac{M : A}{\textbf{inl}^B M : A \vee B}\,\vee I_1 \qquad \dfrac{M : B}{\textbf{inr}^A M : A \vee B}\,\vee I_1$$

$$\dfrac{M : A \vee B \quad \begin{array}{c}\overline{u : A}\\ \vdots \\ N : C\end{array} \quad \begin{array}{c}\overline{v : B}\\ \vdots \\ O : C\end{array}}{\textbf{case}\ M \textbf{of}\ \textbf{inl}\ u \Rightarrow N \mid \textbf{inr}\ v \Rightarrow O : C}\,\vee E$$

$$\dfrac{M : \bot}{\textbf{abort}\ M : C}\,\bot E \qquad \dfrac{}{\langle\rangle : \top}\,\top I$$

FIGURE 3.2: Natural deduction annotated with proof terms.

Moreover, this program will be of a type $A \wedge (A \wedge A \to B) \to B$ and its proof. In Fig.3.3 we provide the deduction of such lambda function.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{u : A \land (A \land A \to B)}}{\texttt{snd}\ \ u : A \land A \to B}\land E_R \qquad \dfrac{\dfrac{}{w : A} \quad \dfrac{}{w : A}}{(w,w) : A \land A}\land I}{\texttt{snd}\ \ u(w,w) : B}\to E}{\lambda w.\texttt{snd}\ \ u(w,w) : A \to B}\to I^w \qquad \dfrac{\dfrac{}{u : A \land (A \land A \to B)}}{\texttt{fst}\ \ u : A}\land E_L}{(\lambda w.\texttt{snd}\ \ u(w,w))\texttt{fst}\ \ u : B}\to E}{\lambda u.((\lambda w.\texttt{snd}\ \ u(w,w))\texttt{fst}\ \ u) : A \land (A \land A \to B) \to B}\to I^u$$

FIGURE 3.3: Natural deduction of the lambda function that proves $A \land (A \land A \to B) \to B$

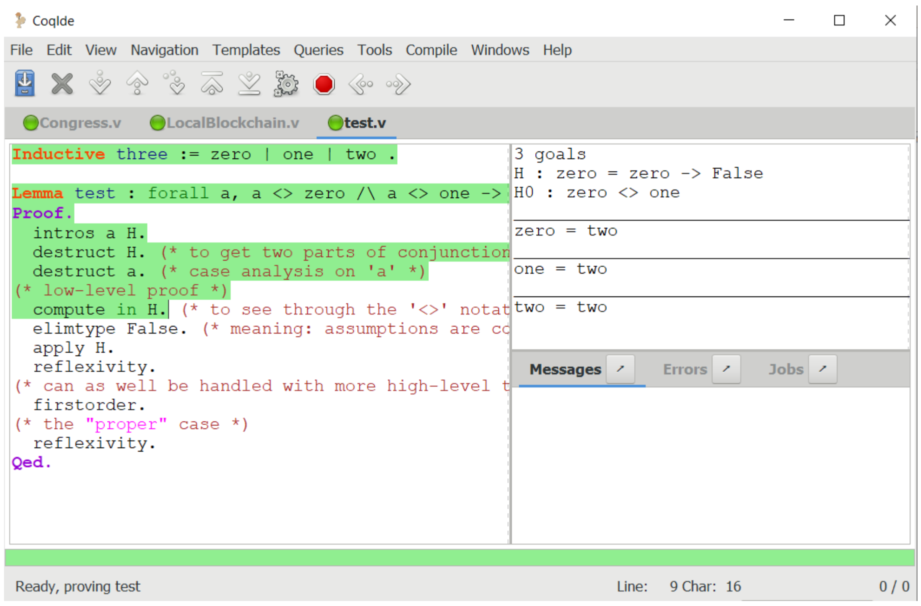
Finally, the resulting lambda function is $\lambda u.((\lambda w.\texttt{snd}\ \ u(w,w))\texttt{fst}\ \ u)$ that has type of the initial proposition. It is worth noting that evaluation of this function on the input data is equivalent to the simplification of the initial proof.

The general logic of the Curry-Howard isomorphism makes it possible to write programs in various functional programming languages, such as Coq, instead of proving theorems in mathematical logic, which greatly facilitates the creation of appropriate formal verification environments. Thus, type theory, lambda calculus, and natural deduction are the primary tools used by any FPMS system.

3.2 FPMS Coq

Coq is a software implementation of the formal proof management system (FPMS). It consists of the Gallina functional programming language and the CoqIDE programming environment. The Gallina language has a wide range of tools for working with mathematical objects of different levels. They are tactics, proofs, theorems, lemmas, predicates, sets, etc. To ensure logical consistency, it is based on the typification of objects. In general, the user has access to the types of these sorts - Prop, Set and Type.

Gallina's mechanism for proving a theorem or lemma is presented through the use of tactics. These are the rules that the user uses to move from one goal to another, eventually reaching the basics of the proposition. In the process of proof, subgoals are created that require processing. Thus, derivation in the Coq environment presents itself as an inverse deduction system. The most commonly used tactics are `intro` and `destruct`. They are analogous to the rules of introduction and elimination from the natural deduction of Gentzen [3].


CoqIde
File Edit View Navigation Templates Queries Tools Compile Windows Help
Congress.v
LocalBlockchain.v
test.v
Messages
Errors
Jobs

```
Inductive three := zero | one | two .

Lemma test : forall a, a <> zero /\ a <> one ->
Proof.
  intros a H.
  destruct H. (* to get two parts of conjunction
  destruct a. (* case analysis on 'a' *)
(* low-level proof *)
  compute in H. (* to see through the '<>' notat
  elimtype False. (* meaning: assumptions are cc
  apply H.
  reflexivity.
(* can as well be handled with more high-level t
  firstorder.
(* the "proper" case *)
  reflexivity.
Qed.
```

```
3 goals
H : zero = zero -> False
H0 : zero <> one
______________________________________
zero = two
______________________________________
one = two
______________________________________
two = two
```

```
Ready, proving test                    Line:  9 Char:  16            0 / 0
```

FIGURE 3.4: CoqIDE theorem proving environment.

Coq is endowed with a minimal system of ready-made types, but the structure of the functional Gallina programming language allows for the flexible creation of new types. In addition to a simple inductive definition through the keyword `Inductive`, it is possible to add more complex types through typed classes, records, and instances. The standard library already has some predefined types, such as natural numbers, booleans, and well-known data structures, lists, hash tables, etc [14]. For example, consider the definition of the inductive type `nat`, which represents the type of natural numbers Fig.3.5.

```
Inductive nat : Type :=
  | O
  | S (n : nat).

Definition pred (n : nat) : nat :=
  match n with
  | O     O
  | S n'    n'
  end.
```

FIGURE 3.5: Definition of the type of the natural numbers and function `pred`

The keyword `Inductive` indicates that it is necessary to define a new inductive object named `nat` and with a colon and indicates that it is defined as a new type. It contains two entities in Coq called constructs, namely `O` and the function `S`, which takes as an input one argument `n` of type `nat`. So, all individuals of type `nat` have the following form:

`O,S(O),S(S(O)),...`

Just as natural numbers are determined in lambda numbers, here we have a zero element `O`, and all subsequent ones are constructed as successors to the previous one, by applying the function `S` to the number.

The `Definition` keyword indicates that we want to define a new function. In the given example, we have function `pred`, which accepts an input of an element of inductive type `nat` and returns another element of type `nat`. This function represents the previous natural number to the given, provided that the zero element has itself as a predecessor. The implementation of such a function is possible through the use of the match construct, which compares their given element with the proposed templates. If the number has the form `O`, return `O`, as agreed. If it has the form `S n'` then return the number to which the function `S` has been applied. Since the inductive definition of `nat` does not provide for other type formats, the function is well defined.

These are just a few of the designs that Gallina provides for use. Since Coq uses typing mechanisms, the possibilities of mathematical type theory are reflected in the environment. The Table 3.2 shows the corresponding constructions in type theory and their analogues with Gallina [8].

Introduction rule	Elimination rule
$\dfrac{A\ \mathtt{true} \qquad B\ \mathtt{true}}{A \wedge B\ \mathtt{true}}\wedge\mathrm{I}$	$\dfrac{A \wedge B\ \mathtt{true}}{A\ \mathtt{true}}\wedge E_L$ $\dfrac{A \wedge B\ \mathtt{true}}{B\ \mathtt{true}}\wedge E_R$
$\dfrac{A\ \mathtt{true}}{A \vee B\ \mathtt{true}}\vee I_L$ $\dfrac{B\ \mathtt{true}}{A \vee B\ \mathtt{true}}\vee I_R$	$\dfrac{A \vee B\ \mathtt{true} \qquad \begin{array}{c}\overline{A\ \mathtt{true}}^{\,u}\\ \vdots\\ C\ \mathtt{true}\end{array} \qquad \begin{array}{c}\overline{B\ \mathtt{true}}^{\,w}\\ \vdots\\ C\ \mathtt{true}\end{array}}{C\ \mathtt{true}} E^{u,w}$
$\dfrac{\begin{array}{c}\overline{A\ \mathtt{true}}^{\,u}\\ \vdots\\ B\ \mathtt{true}\end{array}}{A \to B\ \mathtt{true}} \to I^{u}$	$\dfrac{A \to B\ \mathtt{true} \qquad A\ \mathtt{true}}{B\ \mathtt{true}} \to E$
$\dfrac{\begin{array}{c}\overline{A\ \mathtt{true}}^{\,u}\\ \vdots\\ p\ \mathtt{true}\end{array}}{\neg A\ \mathtt{true}} \neg I^{u,p}$	$\dfrac{A\ \mathtt{true} \qquad \neg A\ \mathtt{true}}{C\ \mathtt{true}} \neg E$
$\dfrac{}{\top\ \mathtt{true}} \top I$	-
-	$\dfrac{\bot\ \mathtt{true}}{C\ \mathtt{true}} \bot E$
$\dfrac{[a/x]A\ \mathtt{true}}{\forall x.A\ \mathtt{true}} \forall I^{a}$	$\dfrac{\forall x.A\ \mathtt{true}}{[t/x]A\ \mathtt{true}} \forall E$
$\dfrac{[t/x]A\ \mathtt{true}}{\exists x.A\ \mathtt{true}} \exists I$	$\dfrac{\exists x.A\ \mathtt{true} \qquad \begin{array}{c}\overline{[a/x]A\ \mathtt{true}}^{\,u}\\ \vdots\\ C\ \mathtt{true}\end{array}}{C\ \mathtt{true}} \exists E^{a,u}$

TABLE 3.1: Gentzen's natural deduction rules

Type theory expressions	Gallina language constructions
A	`A`
$A \to A$	`A -> A`
$\Pi x : A.B$	`forall x :  A, B`
$\Box$	`Type`
$\star$	`Set Prop K : Type`
$(\star \to \star)\alpha$	`(fun x :  K => x -> x) A`
$x : A$	`x :  A`
$\lambda x : A.x$	`fun x :  A => x`
$(\lambda x : A.x)z$	`(fun x :  A => x) z`

TABLE 3.2: Type theoretic notions representation in Gallina

Chapter 4

Related Works

In recent years, much research has been conducted in the field of formal verification. The first steps in the technology of formal verification of the blockchain were made in several scientific papers. We will consider some of them that, in our opinion, have made significant progress in the field of research and the most significant impact on our work.

In the work of Zakrzewski [24] they analyze the structure of the grammar of the programming language for writing smart contracts Solidity, the principle of memory and the execution of contract code. They are abstracted and represent the account in the form of a triplet (b, p, s), where b is a balance of an account, p represents account program and s is its internal storage. Program is represented by the pair $(c, cdefs)$ of contract name and the collection of account contract's definitions. The type for values is an integer type with an upper bound to simulate restricted character of standard programming languages.

An important part is the representation of contract's state and evaluation of its program. In the work they model state as a tuple $\mu = (a, m, \sigma, l^f, l^m)$, where a is an address of the current account, m - memory, σ models network, function local store is l^f and modifier local store l^m. To better model the Solidity principle of operation they introduce these two last functions to have a mapping from identifiers of variables to their values. For l^f, l^m are a variable mappings of the currently executing function and modifier respectively.

$$\begin{array}{rll}
 & \vdash msg, \sigma \Rightarrow \sigma' & \text{transactions} \\
p, q, f & \vdash stmt, \mu \Rightarrow out, \mu' & \text{statements} \\
p & \vdash e, \mu \Rightarrow out, \mu' & \text{expressions} \\
p & \vdash e, \mu \Leftarrow out, \mu' & \text{expressions in lvalue position}
\end{array}$$

FIGURE 4.1: Evaluation judgments from [24]

Evaluation judgments (Fig.4.1) are the main principle of the whole system. They show which parts of it change under what conditions. We see, for example, that during a transaction, the network σ receives a message `msg` and changes its state to σ'. The authors provide proofs of properties for such a model of Ethereum smart contracts and show its drawbacks.

Another important work in the field [11] offers a structural model for describing and modelling Ethereum smart contracts. Their model is very detailed and covers the minor details of smart contracts. Their mathematical simulation method provides the ability to integrate blockchains with both depth-first and breadth-first. We used the fragments and ideas from this work to implement our depth-first model.

The disadvantage of this work is that the authors did not implement the syntactic grammar tree of Solidity source code parsing, as is often the case in similar projects.

This model is fully operational at the level of functional programming languages, so it requires that a program is written in Solidity be integrated into the proposed model in Gallina.

Chapter 5

Model Structure

5.1 General Architecture

Our model is based on the representation of the blockchain, smart contracts, and their implementation based on functional programming paradigms. We were inspired by the model proposed in work [11] because it is general enough to cover our requirements and does not use monads, leading to excessive programming complexity. Instead of representing the constant state of the smart contract in functional programming using monads, this model saves the state in the form of serialized data that can be reproduced at runtime. This serialized data is also used as messages containing transactions in Ethereum. In our approach, we have changed the principle of serialization using third-party libraries. We have identified the main fragments of this model, supplemented them with additional functionality and redesigned the part that does not meet our needs.

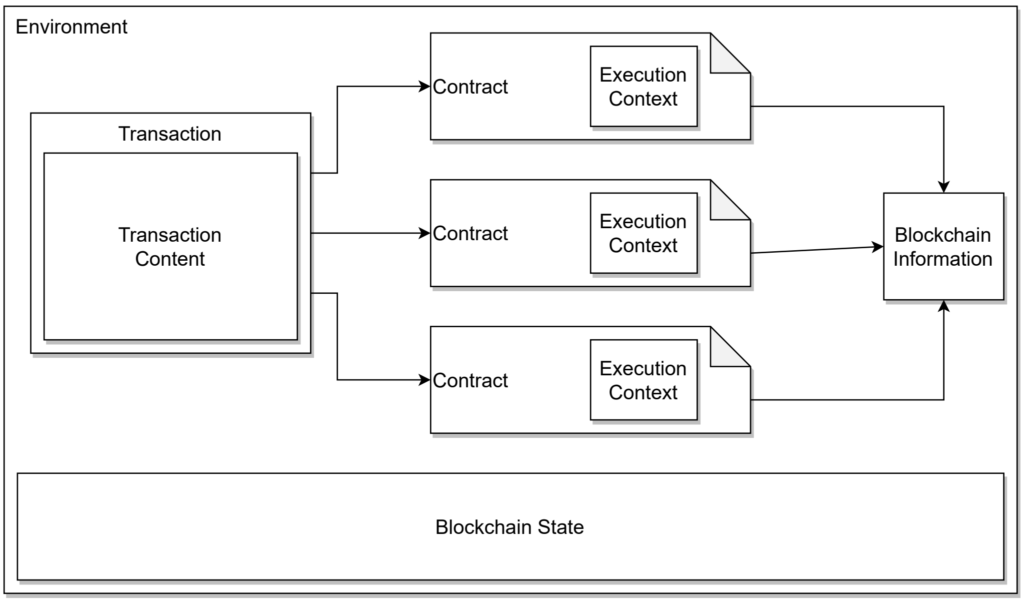


FIGURE 5.1: Model of the blockchain smart contracts

Consider the main components of the model:

- **Blockchain State**: a class that defines the requirements for basic objects in the system, such as addresses, their countability, verification that the address belongs to a person or the contract itself.

- **Blockchain Information**: a class that allows you to get information about the account in the network, its address, information about its type (whether it is a user account or a smart contract). Basic class for working with a model.
- **Environment**: a controlling record that contains the Blockchain Information and Blockchain State classes and allows you to interact with accounts, get information about them for smart cats when executing their code. In other words, it is an interface for the interaction of system elements.
- **Transaction**: a wrapper record that isolates information about the sender of the transaction. Contains the sender's address and the contents of the transaction itself.
- **Transaction Content**: the content of a transaction is defined as an inductive type that has the following constructors: a function that determines the address to which the transaction is sent and the amount of internal currency (bitcoin, ether, gas, etc.) that is laid for this transaction. Function to perform the same action as the previous one, but with a serialized message. Required for creating new contracts and transmitting input to execute their internal program code. Additionally contains functionality to create a smart contract. To do this, the contractors will have to communicate with the blockchain in order to initiate a new contract.
- **Contract**: receives information about the input data and its status during the transaction. Conceptually, the smart contract in our model is a finite state machine, in which the state changes depending on the input data. Software implementation requires that contract functions provide its serialized state and the input from which it will return another state to the network. Importantly, the contract can return a list of other transactions that you must first complete before returning your status. These are the functions and methods that require external interaction with the blockchain network. Only after such a consistent execution of all transactions will our current contract return to its status and the transaction will be considered completed.

```
Record BlockchainInfo := build_bc_info
{
  bc_height : nat;
  bc_slot : nat;
  bc_total_height : nat;
  account_balance : Address -> Amount;
}.
```

FIGURE 5.2: Coq implementation of the `BlockchainInfo` record type-class that represents blockchain status that is accessible for smart contracts.

Fig.5.2 shows the part of the model designed for Gallina. The `BlockchainInfo` record contains information about the blockchain, namely its height, available slot and overall height taking into account the free slots in the chain. These three values are of type `nat`, which is predefined by the Coq medium. The account balance function is of type `Address -> Amount` and is a model for obtaining an account balance at its address.

5.2 Computational Reflection

An important aspect that should be taken into account when designing such systems is the need to implement the principle of computational reflection [1].

Since Coq is based on the principles of intuitionistic logic and natural deduction, in the Gallina programming language, the `bool` and `Prop` types have completely different implementations and by default have no connection between them. Therefore, in order to conveniently prove the theorems, for example, in a situation where it is necessary to analyze the specific values of the truth of statements, it is important to create a mechanism for interchangeable use of these types. This principle is called *computational reflection*.

Computational reflection is an approach in which the values of one type are reflected on the type of statements that are fundamental to the system. This approach is often used in mathematics and logic. One of the best-known examples is Godel's numbering, in which statements are mapped to integer values.

```
Inductive reflect (P : Prop) : bool -> Set :=
  | ReflectT : P -> reflect P true
  | ReflectF : ~ P -> reflect P false.
```

FIGURE 5.3: `Bool.Reflect` type in Coq

The system already has an inductive type, which is used as a common framework to unify the use of computational reflection. As can be seen from the meaning in Fig.5.3, the beginnings of the truth of the statement are reflected in the Boolean type, which corresponds to the intuitive understanding.

```
Lemma bc_address_eq_refl `{BlockchainState} x :
  address_eqb x x = true.
Proof.
 destruct (address_eqb_spec x x).
 auto.
 congruence.
Qed.
```

FIGURE 5.4: `BlockchainState` proof of address equality using computational reflection

One of the uses of computational reflection is to map the statements about the equality of addresses in the blockchain to boolean values. This allows one to easily prove the lemmas on compliance with boolean values. In Fig.5.4 the function `address_eqb_spec` of type `Address -> Address -> bool` after reflection is easily integrated into work with boolean values and proofs.

5.3 External Instruments

The standard Coq comes with a minimum number of types and tools to work with, so we used two external libraries that provide functionality.

- **stdpp**: provides a great number of definitions and lemmas for common data structures such as lists, finite maps, finite sets, and finite multisets. It uses type classes to keep track of common properties of types, like it having decidable equality or being countable or finite [18].

- **Coq-Ceres**: library which is used to encode messages that are used in transactions and smart contract state encoding. S-expressions are uniform representations of structured data. They are an alternative to plain strings as used by Show in Haskell and Debug in Rust for example. S-expressions are more amenable to programmatic consumption, avoiding custom parsers and enabling flexible formatting strategies [2].

5.4 Safe Remote Purchase Contract

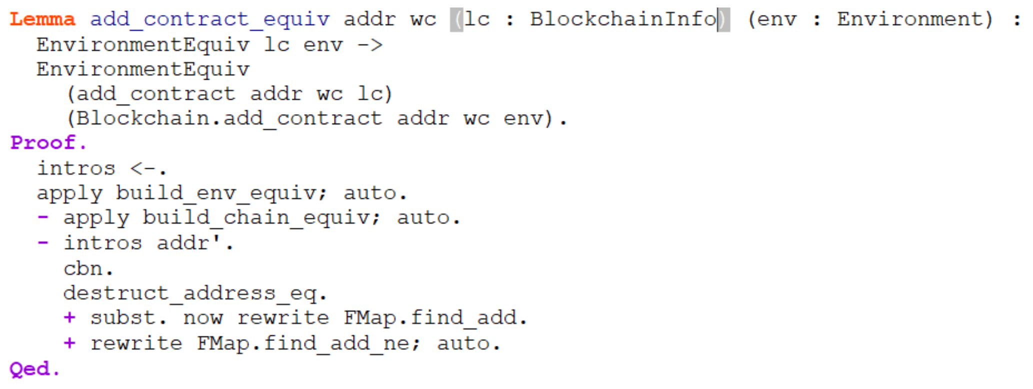
One of the goals of our work is to test hypotheses about existing known types of Solidity smart contracts. One of these is Safe Remote Purchase Contract or SRPC.

This contract is used for online purchase where both the seller and the buyer deposit ethers twice the value of the goods into the contract and can only get their de- posits back when the buyer confirms receipt of the goods. The seller is disincentivized from withhold- ing the goods (not shipping them), and the buyer is disincentivized from not confirming receipt, because in doing so they will lose more than the value of the goods.

Chapter 6

Results and Conclusion

As a result, we have a system that still requires refinement, but already simulates the basic functionality of the blockchain. The model is not complete, as it omits many aspects of true blockchain systems, such as hashing and other levels that go beyond the level of vionization and applications.



```
Lemma add_contract_equiv addr wc (lc : BlockchainInfo) (env : Environment) :
  EnvironmentEquiv lc env ->
  EnvironmentEquiv
    (add_contract addr wc lc)
    (Blockchain.add_contract addr wc env).
Proof.
  intros <-.
  apply build_env_equiv; auto.
  - apply build_chain_equiv; auto.
  - intros addr'.
    cbn.
    destruct_address_eq.
    + subst. now rewrite FMap.find_add.
    + rewrite FMap.find_add_ne; auto.
Qed.
```

FIGURE 6.1: Lemma with proof of equivalence of blockchains

As an example, the code diagram shows a lemma that reflects the fact that if the state changes in two blockchains by the same amount, their equivalence will not change. This requires proof with additional ancillary lemmas, as well as the rewriting mechanism that we implemented earlier.

This study required a verbal theoretical study of the sources, as well as fungal practice with a system of proof of Coq's theorems. We continue to work on this work to complete it, but a wide layer of work in design, research and programming has been done.

Bibliography

[1] Yves Bertot. *Coq in a Hurry. 3rd cycle*. 2016.

[2] *Coq-Ceres documentation*. `github.com/Lysxia/coq-ceres`.

[3] *Coq documentation*. `coq.inria.fr/documentation`.

[4] Edsger Wybe Dijkstra. *A Discipline of Programming*. 1st ed. Prentice-Hall, Inc., 1976.

[5] Wolfgang Thomas H.-D. Ebbinghaus J. Flum. *Mathematical logic*. 1st ed. Springer New York, 1996.

[6] John Harrison. *Formal verification*. Lecture notes from Marktoberdorf. 2010.

[7] Macor Jackson. *A brief introduction to type theory and the univalence theorem*.

[8] Vasyl Lenko. “Methods and tools for personal knowledge management in intelligent systems”. In: (2020).

[9] Jürgen Trinks Maria Fürst. *Philosophie*. 2nd ed. Dukh i litera, 2019.

[10] Pradhan Pattanayak Sanjeev Verma Vignesh Kalyanaraman Michael Crosby Nachiappan. *BlockChain Technology Beyond Bitcoin*. Sutardja Center for Entrepreneurship Technology Technical Report. 2015.

[11] Jakob Botsch Nielsen and Bas Spitters. “Smart Contract Interactions in Coq”. In: (2019). DOI: `10.48550/ARXIV.1911.04732`. URL: `https://arxiv.org/abs/1911.04732`.

[12] Sebastián E. Peyrott. *An Introduction to Ethereum and Smart Contracts*. Auth0 Inc. Version 0.1.0. 2017.

[13] Frank Pfenning. *Automated Theorem Proving*. Material for the course Automated Theorem Proving at Carnegie Mellon University, Fall 1999, revised Spring 2004. This includes revised excerpts from the course notes on Linear Logic (Spring 1998) and Computation and Deduction (Spring 1997). 2004.

[14] Benjamin C. Pierce et al. *Software Foundations*. Version 5.0. http://www.cis.upenn.edu/ bcpierce/sf. Electronic textbook, 2017.

[15] Giselle Reis. *Curry-Howard correspondence*. Lecture 04, Constructive Logic (15-317).

[16] Raul Rojas. *A Tutorial Introduction to the Lambda Calculus*. 2015. DOI: `10.48550/ARXIV.1503.09060`. URL: `https://arxiv.org/abs/1503.09060`.

[17] *Solidity documentation*. `docs.soliditylang.org/en/v0.8.14/`.

[18] *Stdpp documentation*. `gitlab.mpi-sws.org/iris/stdpp`.

[19] Nimesh Prakash Vivek Acharya Anand Eswararao Yerrapati. *Oracle Blockchain Quick Start Guide. A practical approach to implementing blockchain in your enterprise*. 1st ed. Packt Publishing Ltd, 2019.

[20] Philip Wadler. “Propositions as Types”. In: *Commun. ACM* 58.12 (2015), 75–84. ISSN: 0001-0782. DOI: `10.1145/2699407`. URL: `https://doi.org/10.1145/2699407`.

[21] Wenbo Wang et al. “A Survey on Consensus Mechanisms and Mining Strategy Management in Blockchain Networks”. In: *IEEE Access* 7 (2019), pp. 22328–22370. DOI: `10.1109/ACCESS.2019.2896108`.

[22] Alfred North Whitehead and Bertrand Russell. *Principia Mathematica*. Cambridge University Press, 1925–1927.

[23] Puliutin E. Yershov U. *Mathematical logic*. 1st ed. Moskva "Nauka", 1979.

[24] Jakub Zakrzewski. “Towards Verification of Ethereum Smart Contracts: A Formalization of Core of Solidity.” In: *VSTTE*. Ed. by Ruzica Piskac and Philipp Rümmer. Vol. 11294. Lecture Notes in Computer Science. Springer, 2018, pp. 229–247. ISBN: 978-3-030-03592-1. URL: `http://dblp.uni-trier.de/db/conf/vstte/vstte2018.html#Zakrzewski18`.