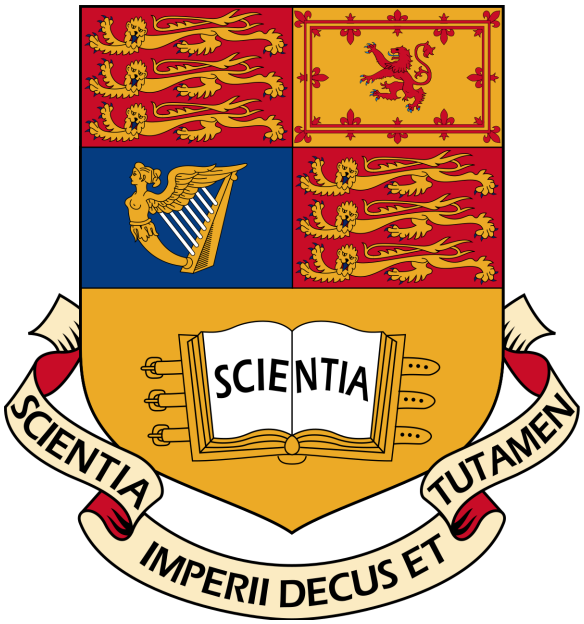
Imperial College London

Department of Computing



Nexus: A Domain-Specific Language for Financial Smart Contracts

by

Noah-Vincenz Noeh (NVN)

Submitted in partial fulfilment of the requirements for the MSc Degree in Advanced Computing of Imperial College London

September 2019

Abstract

Financial institutions rely more and more on blockchain technology and financial smart contracts are becoming more prominent as time goes by. Financial smart contracts allow for efficient automation of contract execution and their contractual terms, while providing the cost and security benefits of blockchain technology.

The most prominent platform for smart contracts is Ethereum. Ethereum smart contracts are implemented in the Solidity programming language. Because of the design and underlying complexity of this high-level language, which allows for many unsafe programming patterns, large amounts of Ether (the cryptocurrency used in Ethereum) have been locked and stolen. Generally speaking, smart contract development in Solidity is non-trivial and highly prone to a number of security vulnerabilities.

As a result of the above, we introduce Nexus: a high-level domain-specific programming language for writing safer financial smart contracts. This language is based on previous work in the financial engineering industry conducted by Simon Peyton Jones, Jean-Marc Eber, and Julian Seward [1]. Nexus is a high-level programming language that is straightforward to use, efficient, and easily portable to any underlying blockchain. We also present an accompanying web application that allows users to easily compose smart contracts in Nexus, while handling their compilation in the background. The interface of the web application also enables users to interact with Nexus smart contracts, evaluate these, as well as visualise their evolution. This allows for smart contract transparency, effectively enabling users to reason more easily about financial smart contracts.

Acknowledgements

I would like to thank:

- Professor Susan Eisenbach and Professor Sophia Drossopoulou, for guiding me throughout this project with their advice and knowledge, making time for me, and without whom I would not have managed to achieve the goals of this project,
- Dr Arthur Gervais, for introducing me to the blockchain world and passing on his knowledge with his exciting Distributed Ledgers lectures taught at Imperial College London,
- The Department of Computing at Imperial College London, for making my time during my MSc in Advanced Computing an invaluable experience,
- My friends, for supporting me throughout this journey and providing me with valuable feedback for the system,
- And lastly, my family, for being my role models, having allowed me to come this far, and always providing me with emotional support.

Contents

Chapter 1

Introduction

This project aims to create a high-level domain-specific language for financial smart contracts on the blockchain. Nowadays, in order to use the newest cutting-edge technology, users are usually required to possess a set of technical skills. Self-executing financial smart contracts is currently one of the hot topics in both, the technology and finance industry. For the first time in history, the World Bank and Commonwealth Bank of Australia have announced their blockchain-enabled government bond [2] and Spanish banking group BBVA signed their first blockchain-based loan in July 2018 [3]. Other organisations in the financial industry have raised awareness of these events and blockchain technology is more prominent in the industry than ever before. As stated by Partz in her Cointelegraph news article [3], Blockchain technology can be leveraged for numerous operations in the financial industry, allowing to enhance efficiency, transparency and traceability.

Nexus is a domain-specific language for financial smart contracts that brings together blockchain technology and finance, allowing individuals to connect and interact with each other via blockchain transactions securely, with ease and at little cost. The ultimate goal of this project is to have a user-friendly web interface that any user, with little technical background or coding ability, can interact with by supplying simple smart contract text input written in the Nexus language. The supplied input will be parsed, verified, broken down into its syntactical components, evaluated and eventually transformed in order to create a smart contract instance on the blockchain. We also want to allow users to understand more about their transactions and general market activity, while maintaining the privacy of all involved parties.

Enormous amounts of Ether — the most commonly used cryptocurrency for smart contracts and the second largest digital currency in the world — have been locked and stolen as a result of programming language coding vulnerabilities present with Ethereum's language for developing smart contracts: Solidity. With Nexus, end users are no longer required to write smart contract code themselves in order to make a financial transaction on the blockchain, because the code to be deployed is pre-defined and inalterable, significantly reducing these threats. The known to be lengthy and time-consuming process of deploying carefully designed smart contract code, which usually takes at least a week to develop and additional weeks to test, is made obsolete by this application. Therefore, the potential impact that this research can have on the

technological and financial sector, as well as on our everyday life, is undeniable.

A fundamental part of this application involves the accurate parsing of the user input and its compilation into a programming language that defines a set of combinators. The key idea of these so-called combinators is to summarise complex contracts into small sets of simpler contracts as a single formal description, allowing users to define smart contracts in a declarative way. In the finance industry, this enables automation of a number of different process, including back office contract execution, risk analysis optimisation, simulations and graphical representations [1].

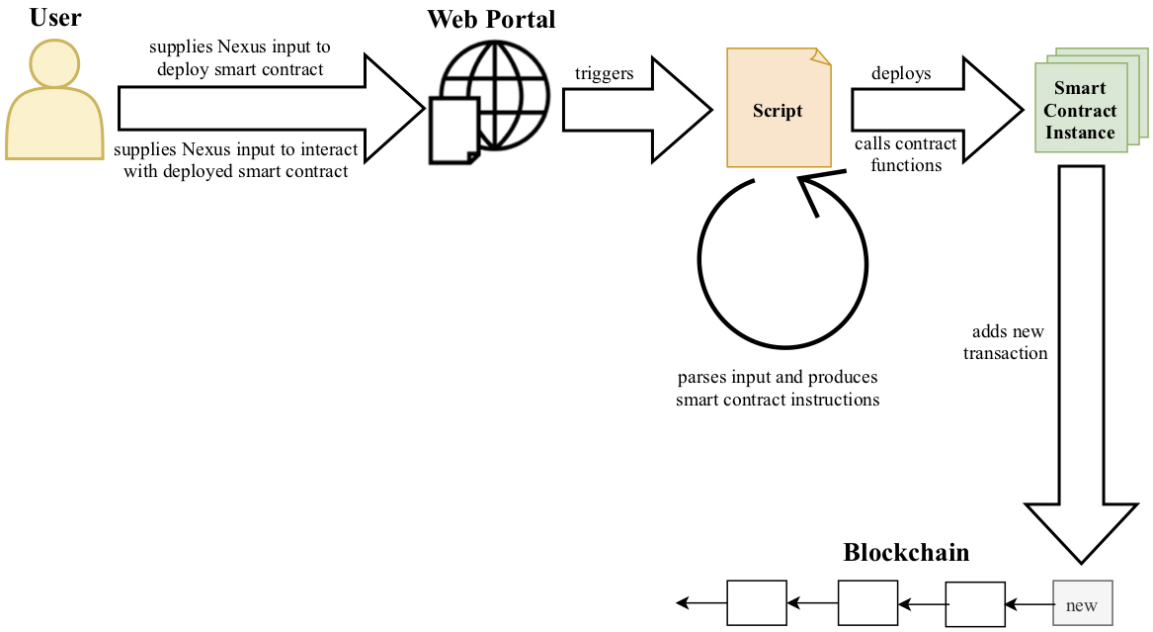


Figure 1.1: System Outline

The above diagram outlines the general structure of the system. A user provides input over a web portal, which is parsed and compiled into smart contract code that will eventually be added to a blockchain. Chapter 6 will provide more detail on which technologies were chosen for the system implementation, how the system was architecturally designed, the tasks of different system components and how these interact with each other.

The following chapter of this paper will introduce and elaborate on the key concepts of this paper, involving blockchain and specifically the Ethereum and Libra blockchains, Smart Contracts and Combinators. Chapter 3 will analyse literature that was reviewed prior to the development stage of this project. This will outline existing progress in this field, providing evidence that the research investigated for this project represents

a hot topic in the technological and financial world. Chapter 4 will define the requirements and specification that were needed for the success of this project, describing which tasks the software must be able to perform and what it should not do. It will also outline the tasks of different system components. This is followed by chapter 6, elaborating on how the system was implemented. The following chapter, namely chapter 7, will explain different software testing approaches that were taken to allow for a successful project. Moreover, this chapter will describe how the system was tested to satisfy the requirements specified in previous sections. Chapter 8 will evaluate our implementation in terms of our requirements, as well as the system's functionality, efficiency and usability, and will compare these to existing solutions. Finally, this paper will conclude in chapter 9, conveying the potential of Nexus, introducing future ideas involving technical improvements and feature additions that were simply beyond the scope of this this time-bounded implementation, and finally, reflecting on the project outcome.

Chapter 2

Background

2.1 Blockchain

The blockchain was first commercialised by Satoshi Nakamoto, the inventor of Bitcoin, the world's first and most well-known cryptocurrency. Satoshi Nakamoto, however, is said to be only a pseudonym and no one really knows the real identity of the Bitcoin inventor. In his Bitcoin paper [4] published in 2008 he first explained the blockchain concept. In 2009 the Bitcoin network was launched with Nakamoto mining the first block of the chain, known as the genesis block. Although Nakamoto was the first 'person' to publish the blockchain concept with his paper, similar ideas had been found in this research area prior to these efforts. Haber and Stornetta [5] discuss research regarding the timestamping of digital documents, introducing the concepts of irreversibility and immutability of records, whereas Nick Szabo first introduced the idea of leveraging decentralised ledgers for the use of smart contracts in 1994 [6].

A blockchain can be seen as a publicly distributed database, often referred to as the ledger, that stores records in the form of blocks. Instead of having data being stored at some central location, data is distributed among all nodes within the network. Generally, there are different kinds of nodes in a blockchain network. The two main types comprise nodes that maintain the entire record of the blockchain, referred to as full nodes in Bitcoin, and all other nodes participating in the network, referred to as lightweight nodes in Bitcoin. Each block in the chain is a collection of processed transactions and is linked to its previous block via a hash. When a new transaction is made on the network, it needs to be verified by all existing nodes in the network prior to being added to the blockchain — this creates a degree of security and reliability through irreversibility and immutability.

There are two different classifications of blockchains. Permissionless blockchains, such as the Ethereum (see section 2.4) and Bitcoin blockchains, are the more popular kind of blockchains and allow any unknown user to make transactions, create a node and start mining. Permissioned blockchains, such as Libra (see section 2.5), on the other hand, do not allow users to freely join the network and often rely on trusted providers to enable nodes to join, hence requiring some sort of central control. Therefore, it can be stated that permissioned blockchains are more of a centralised system as com-

pared to their permissionless counterparts. While this project will lay focus on the use of permissionless blockchains, it would be interesting to implement a domain-specific language for smart contracts based on both types of blockchains and then draw a comparison between the two solutions in terms of security, performance and privacy.

2.2 Consensus: PoW vs PoS

Every blockchain relies on some sort of consensus algorithm. A consensus algorithm can be defined as a mechanism that nodes in a network rely on in order to agree on a common decision, or 'reach consensus', on the current system state. A consensus algorithm ensures that the global ledger state is up-to-date and correct, in other words all transactions are valid, allowing for a public agreement on the order of transactions. There are two main types of blockchain consenus algorithms: Proof of Work (PoW) and Proof of Stake (PoS).

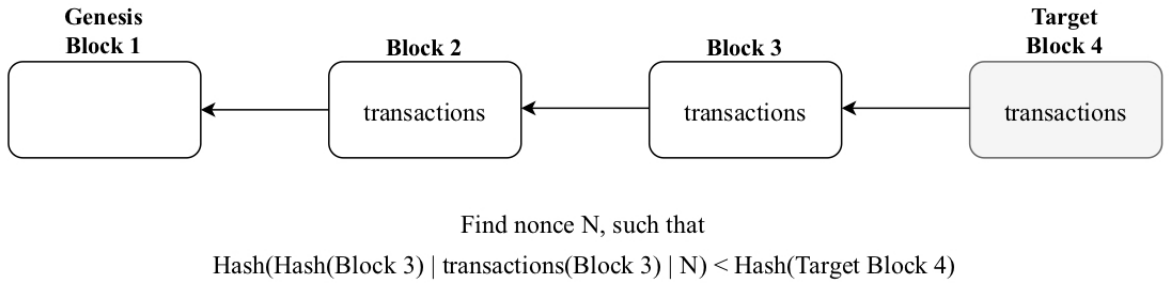


Figure 2.1: PoW Blockchain Mining

In PoW systems, when a transaction is made on the network, it needs to be verified by all existing nodes in the network. Nodes that mine the next block in the chain of such a system, do so by bundling together all transactions that are currently available to them and not included in the main chain. A template block is formed. This block is successfully mined when all the transactions included are verified and the mining node has found a random word, referred to as nonce, which satisfies the difficulty of the target block when being hashed together with the current block's data. This difficulty is equivalent to the number of leading 0s in the target hash, as illustrated in figure 2.1. Finding such a nonce N is a mathematically hard problem to solve and the currently best approach known for solving this is to bruteforce all hashes, that is randomly guessing hashes until eventually finding the solution. Naturally, the more leading 0s the target hash contains, the greater the difficulty to find such an N, and hence the longer it will take to generate the next block in the chain. The probability of successfully mining the next block in PoW is proportional to the computational power used to mine that block. This builds the security foundation of any PoW blockchain — malicious nodes cannot simply propagate some new block, as this block would not conform to the blockchain's history. For instance, if a malicious node wanted to change a past transaction contained inside some existing block B_{i-x}, where B_i represents the most recently mined block in the main chain, they would not only need to find a valid nonce for B_{i-x}, but also for all the following x blocks up to and including B_i,

and if successful, be the first one to find a valid N for B_{i+1} in order to push the updated blockchain state to the network. The above assumes that during this process the remaining nodes in the network have not made any progress in finding N for B_i. However, in a decentralised network with honest nodes this scenario is highly unlikely, because all other nodes in the network are working on finding the nonce for the current block in the main chain while the malicious node is trying to catch up to this chain. Honest nodes are defined as network nodes that remain available and do not behave maliciously. Nakamoto [4] discovered that if the probability of an honest node finding a block is less than or equal to the probability of a malicious node finding a block, the malicious node will catch up to the main chain. Otherwise, the probability of the malicious node catching up decreases exponentially with every newly added block on the main chain — something referred to as transaction finality, which is beyond the scope of this paper. This effectively makes such an attack infeasible, unless the malicious node possesses over 51% of the entire network computing power, effectively being in control of the network (referred to as 51% attack). By enforcing nodes to solve this complex cryptographic puzzle in order to mine the next block, PoW consensus systems rely on energy and power intensive mining. This has led to problems particularly in the Bitcoin network, where mining is now dominated by so-called Application-Specific Integrated Circuits (ASICs). ASICs can be seen as hardware designed for a specific, computationally expensive use rather than general purpose [7]. There are ASICs designed specifically for Bitcoin mining that are particularly powerful in computing the required sha256 hashes and outperform any other machine in doing so. This effectively makes using an ASIC the only profitable way of mining in Bitcoin as a single node.

If a correct block has been found and all transactions included in the block are valid, the block will be propagated within the network and it will be confirmed and adapted by all honest and non-defective nodes in the network. Mining continues on the next block. The node that successfully manages to mine a block is rewarded with the sum of all mining fees that nodes included in their transactions. In Bitcoin this is added on top of the fixed block reward for the block that was mined. The generation of a new block in Bitcoin takes on average 10 minutes, whereas on the Ethereum network this takes between 10 and 20 seconds.

In PoS systems, on the other hand, miners are replaced by validator nodes and the concept of block mining is lost. In ordinary PoS systems the probability of validating and creating a new block in the chain is proportional to the relative share the node owns of the currency. Therefore, this probability is deterministic and fully dependent on how much stake the validator node holds, or in other words, how many coins they possess. In order to validate a block in a PoS system, validators stake coins in their wallet to place a “bet” on some block. If the block on which a bet was placed ends up being added to the chain, the validating node is rewarded with an amount of coins proportionate to their bet. In PoS there is no block reward, so the validator purely benefits from the

transaction fees in the network. Since there is no mining in PoS systems, such systems are known to place lower loads on the blockchain, resulting in reduced power and energy consumption, and therefore represent a more sustainable solution to consensus algorithms. Furthermore, PoS systems often imply greater currency health, as nodes have to own coins of the currency, stake coins in their wallet and actively "participate" in the network in order to successfully validate blocks and gain rewards.

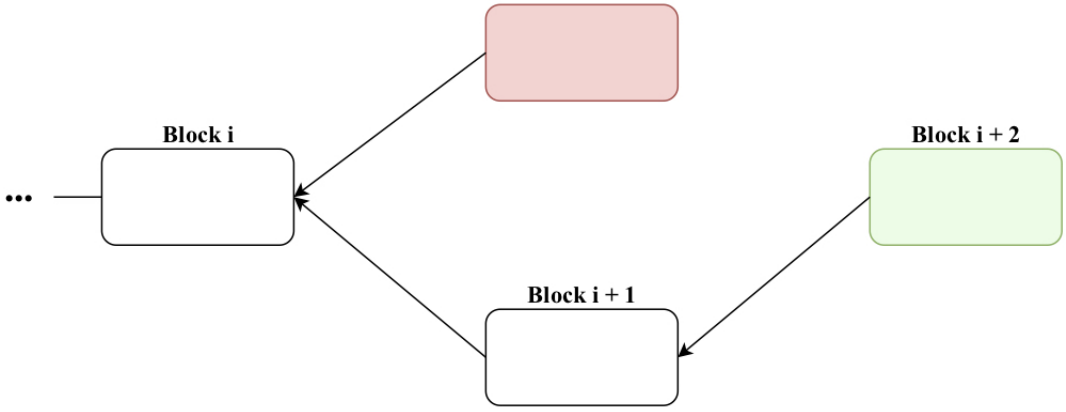


Figure 2.2: Blockchain Fork

A fork occurs when at one instance in time there are multiple newly added valid blocks and there exists a disagreement on which block to include in the main chain. This scenario may arise with both classifications of blockchain systems. The blockchain will fork at this point into two (or more) branches and miners can freely pick which branch to mine on. This implies that multiple blockchain states may coexist at one instance in time. Whichever branch succeeds first in mining the next block in the chain will be merged back to be the main chain, blocks will be updated, and nodes that validated blocks in the included branch will be paid with the mining rewards. In the diagram above, two valid blocks were found for *Block i+1* and the chain was forked into two branches at that point. Nodes that were mining on the red-shaded block will have to adapt the resulting chain of blocks. This scenario leads to an undesirable cost for nodes in PoW systems that pick a branch that does not end up being included, since these nodes waste valuable resources and time on mining. PoS mining, on the other hand, is effortless, meaning PoS nodes can freely build on every fork.

Although the compared set of mechanisms comprises two completely different types of consensus algorithm implementations, the long-term benefits of PoS over PoW systems are undeniable, which is why many modern blockchains rely on PoS. Other consensus algorithms include Delegated Proof of Stake (DPoS), Proof of Authority (PoA) and Federated Byzantine Agreement (FBA).

2.3 Smart Contracts

Originally, Bitcoin solely involved simple payments between two parties. In recent years however, new applications of blockchain technology have emerged — one of them being smart contracts. Blockchain users started to require transactions to execute only when certain conditions are met and wanted to implement their own, custom-defined transactions. This is when smart contracts started to become popular. Smart contracts allow user-defined code on the blockchain to validate contractual terms, store cryptocurrency asset balances, as well as transfer such assets. "A smart contract is a computerized transaction protocol that executes the terms of a contract. The general objectives of smart contract design are to satisfy common contractual conditions (such as payment terms, liens, confidentiality, and even enforcement), minimize exceptions both malicious and accidental, and minimize the need for trusted intermediaries" [6]. Smart contracts allow one to exchange anything of value without having to rely on a centralised entity such as a bank. Szabo compares the concept of a smart contract to the one of a vending machine. The user inserts some kind of input of value and the vending machine produces an output based on whether a set of if-then conditions are met. Smart contracts are transformed into code that is stored on the blockchain, which again benefits from security through immmutability, which in turn creates trust and a reliable source of backup. Smart contracts are often referred to as self-executing contracts, as they not only define the rules and penalties of a contract, but automatically enforce these upon a triggering event, for example when a certain date and time has been reached. This results in increased autonomy. Furthermore, smart contracts are capable of enhancing transaction speeds since no central entity is involved, increasing savings due to reduced service and reduced international payment costs, as well as decreasing the need for a physical currency and general paperwork. The benefits smart contracts can deliver in different applications are undeniable and there is a great potential for technological progress in this area of research. Nowadays, there already exist many different smart contract use cases ranging from supply chain management, where companies track where their goods originate from [8], to healthcare and mortgage applications.

This paper presents a domain-specific language for financial smart contracts on the blockchain, allowing two parties to send transactions between in each other at ease, while taking advantage of the benefits provided by these underlying technologies. Figure 2.3 represents an example of a simple financial contract between two parties taken from Peyton Jones' paper.

D_1 Both of:

- D_{11} Receive £100 on 29 Jan 2001.
- D_{12} Pay £105 on 1 Feb 2002.

D_2 An option exercisable on 15 Dec 2000 to choose one of:

- D_{21} Both of:
  - D_{211} Receive £100 on 29 Jan 2001.
  - D_{212} Pay £106 on 1 Feb 2002.
- D_{22} Both of:
  - D_{221} Receive £100 on 29 Jan 2001.
  - D_{222} Pay £112 on 1 Feb 2003.

Figure 2.3: A Basic Financial Smart Contract [1]

Every contract involves two parties. Firstly, the holder of the contract, who can be seen as the owner of the contract that receives the payments written out in the contract. The owner of a contract is the party with the rights of the contract, involving the rights to decide on contract choices and acquirement. The second party in a contract is defined as the contract counter-party who pays the owner of the contract. In the contract shown in figure 2.3 the owner has the right to choose on 30 June 2000 between D_1 and D_2, where each gives the owner further options to choose from. This is a perfect example of a more complex financial contract being broken down into and composed of multiple simpler contracts D_1, D_{11}, D_{12}, D_2 et cetera. Throughout this paper we will refer to compositional contracts like the contract demonstrated above as supercontracts, and non-compositional contracts like D_{21} as lowest-level contracts (or lowest-level subcontracts).

Originally, the implementation of smart contracts and other applications on top of the Bitcoin protocol was not feasible due to the fact that Bitcoin limits the number of operations one can execute with transaction scripts in order to minimise CPU/memory usage, making program execution time and memory bounded by the size of the program being executed. Bitcoin was not designed to allow for complex programs such as programs that may loop forever, hence loops are not implemented in its scripting language. This makes the underlying Bitcoin scripting language *Script* always have a deterministic state and thus non-Turing-complete. Generally speaking, for a programming language to be labelled as Turing-complete it must allow for reading and writing to some storage, for instance via variables, as well as implement conditional jumps or repetitions, for instance via loops or goto jumps.

As time went by and smart contracts started to gain popularity in the distributed

ledgers world, new blockchain technologies were invented to particularly support such applications on top of the blockchain layer. In 2017, the Particl Foundation published Particl.io [9], a privacy-focused and decentralised ecosystem that allows decentralised apps (DApps) and smart contracts to be implemented on top of the Bitcoin protocol. Unlike other efforts made into this direction, Particl.io builds on top of the existing Bitcoin protocol, aiming to enhance the strong foundations of the most well-known cryptocurrency, instead of just forking off the Bitcoin protocol to create a new cryptocurrency [10]. Particl.io is able to do so by using a scheme referred to as Mutually Assured Destruction (MAD) escrow. Fundamentally, MAD escrow relies on a deposit of a certain amount of funds by two parties that acts as a handshake between them, releasing the funds only on agreement and satisfaction of contractual terms. Although Particl.io successfully enables users to develop smart contracts and build DApps in Bitcoin, it is a complex implementation that offers only limited smart contract functionality as compared to smart contracts implemented in Ethereum.

2.4 Ethereum

The Ethereum blockchain was first announced in 2013 by cryptocurrency researcher and programmer Vitalik Buterin and is now known as the second most prominent blockchain with regards to traffic, global volume and mentions in the web community [11]. It is a permissionless blockchain that uses a digital currency referred to as Ether. Solidity, Ethereum's high-level programming language used to create smart contracts on the Ethereum blockchain, implements loops, making the language Turing-complete and allowing the Ethereum Virtual Machine (EVM) to compute almost anything given enough resources. Although these loop constructs may run infinitely, this is not an issue in Ethereum as long as transactors pay the required transaction fee per computational step. A simple Solidity program containing a loop construct, having the same syntax as loops in prominent programming languages like C, C++ and JavaScript, is presented below.

```
address[] public addressArray;

// add addresses to the array
addressArray.push(address1);
addressArray.push(address2);
addressArray.push(address3);
addressArray.push(address4);

// 4
uint lengthOfArray = addressArray.length;

// loop through the array
for (uint i = 0; i < lengthOfArray; i++) {

     // do something

}
```

Listing 2.1: A Simple Loop Construct in Solidity

Turing-completeness, together with value-awareness, blockchain-awareness and state, that were added to the Bitcoin blockchain to create Ethereum, effectively allow users to write complex smart contracts with arbitrary rules, custom-defined DApps, assets that represent the ownership of a physical device (smart property), custom currencies (coloured coins) or domain names [12].

2.4.1 The Ethereum Blockchain

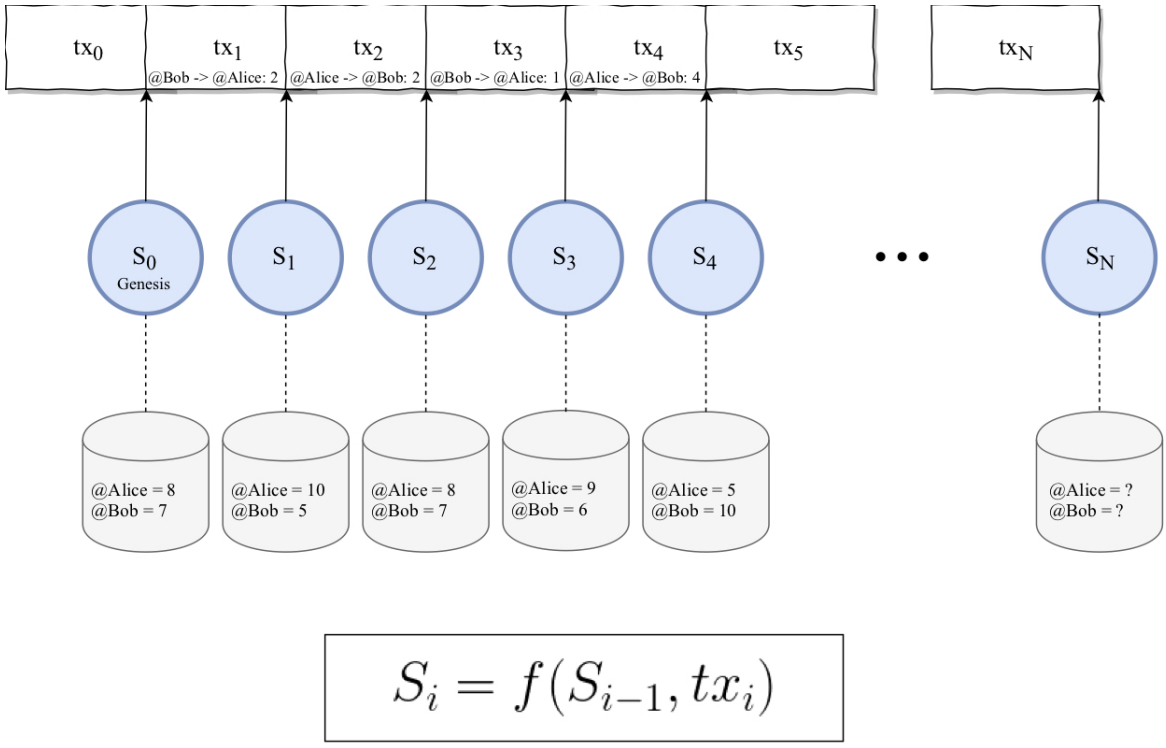


Figure 2.4: The Ethereum Blockchain

The Ethereum blockchain can be seen as a transaction-based state machine, keeping track of the current state (or version) S_i of the ledger and executing transactions to increment that state to S_{i+1} [13]. The current state of the blockchain is also referred to as the block state that the EVM keeps track of — a global state maintaining all accounts, including their balances and storage. For the sake of simplicity, in the figure above blockchain states are only represented by the number of Ether present at each 20-byte account address. The first state in the blockchain, S_0, is referred to as the genesis state, which defines the key components of the ledger via the first transaction in the blockchain, tx_0. Every valid new transaction, tx_i, updates the current state to S_i. For example, at state S_2 Bob's account balance is 7 and Alice's balance is 9 Ether. Bob then sends Alice 1 Ether, which updates Bob and Alice's account balances to 6 and 9 respectively, as well as the current state to S_3.

States are stored in modified Merkle trees, referred to as Patricia trees, as mappings between Ethereum addresses and Ethereum account states. Every account state includes a nonce, representing the number of transactions sent from the account's address

(or if the account is associated with contract code then this represents the number of contract creations made by the account) and practically preventing double spending of transactions. Account states additionally include the account's Ether balance, a hash of the storage contents of the account, and a hash of the EVM code of the account (this is what is eventually executed by the EVM).

Transactions contain the recipient's address, a signature that identifies the sender, the amount of Ether to transfer, optional data, the maximum gas the transaction is allowed to consume before being rejected (21,000 units of gas is the standard gas limit for regular transactions), and lastly, the gas price, which represents the transaction fee the sender pays. Fees in Ethereum ensure that users pay the proportionate amount of Ether for every computation, bandwidth and storage they consume, effectively preventing potential DOS attacks arising from Turing-completeness. When someone buys petrol at a petrol station they pay a certain price in the local currency for the number of units of petrol purchased, let us say £1.40 per litre, whereas in Ethereum one pays a certain price in Wei, Ethereum's smallest subdenomination of Ether (like Satoshi in Bitcoin), for the number of units consumed by the transaction — a unit in Ethereum is referred to as gas. A single computational step is typically equivalent to 1 gas. Transactors freely specify the gas price, however, the higher the gas price, the faster the transaction will be processed, because naturally, miners prefer transactions with higher gas prices. Additionally, miners may ignore transactions with too low a gas price, as there would be little incentive for them to spend their computational resources on such a transaction. There are two types of transactions. Firstly, transactions that result in message calls, allowing for messages to be sent to a contract via the *call()* interface. Secondly, those transactions associated with contract code for the creation of a new contract. The latter type of transactions include an additional transaction field containing the EVM byte code to be executed at contract creation. This code is executed once only.

Transactions are gathered into blocks, each linked to its previous block via a cryptographic hash as described in section 2.1. While Bitcoin enforces a 1 Megabyte (MB) block limit, there is no limit on how much can be stored in Ethereum blocks. However, Ethereum implements a gas limit per block. Bitcoin blocks only maintain a copy of all transactions, whereas blocks in Ethereum maintain the list of transactions together with the most recent state [12]. In order to efficiently store the current state with each block, Ethereum stores states in so-called Patricia trees, modified Merkle trees that allow nodes to not only be updated efficiently, but also to be inserted and deleted efficiently. These trees use pointers in the form of subtree hashes to enable efficient referencing of data. Since all state information is stored within the most recently mined block, the entire blockchain history is no longer required to be stored, allowing full nodes in Ethereum to store only the state, thus significantly reducing the space used [12].

At any given time i, the state of the blockchain S_i can be obtained from a determin-

istic execution function f of the previous state S_{i-1} and the previous transaction tx_i. This function is also referred to as the *Apply* function in Ethereum. The process for the state transition function f consists of the following steps. Firstly, the sender's signature is verified and it is checked whether the transaction is well-formed. A valid transaction includes a nonce that matches the nonce in the sender's account. Secondly, the EVM calculates the transaction fee by multiplying the maximum gas with the gas price and subtracts the result plus the transaction amount from the sender's account balance while incrementing the sender's nonce. Following this, the transaction amount is transferred from the sender's account to the recipient account and their balances are updated. In the case when the receiving account is associated with a contract, the contract's code is executed instead [12].

The above assumes that every step of the procedure succeeds without any disruptions. Such a disruption may occur for instance if the sender does not supply enough gas for the transaction. In this case the EVM throws an out-of-gas exception at the given time of the process, reverting all changes and terminating the transaction. However, the consumption of the transaction fees cannot be reverted and this amount is paid to the account that attempted to mine the failed transaction [12]. Hence, unlike in Bitcoin, execution may be halted abruptly due to a thrown exception. Therefore, Ethereum may not be Turing-complete. This, however, depends on the definition of Turing-completeness, which is why Wood [13] labels Ethereum as quasi-Turing-complete only, since computation is "intrinsically bounded by gas".

2.4.1.1 PoW Consensus

The Ethereum protocol relies on a PoW consensus mechanism where mining nodes dedicate effort or work towards a block and successfully mine only after showing a cryptographically secure proof, as described in section 2.2. The goal of this mechanism is to reach agreement on the path from the root, genesis block, to the leaf, or most recent block in the chain. This protocol is based on a simplified version of the GHOST protocol [14] that aims to accelerate Bitcoin's underlying protocol and alleviate the issue of ASIC-dominant mining. This is achieved by leveraging memory hardness and creating a design that does not allow memory to be used in parallel to discover multiple nonces simultaneously, effectively making it infeasible to implement on ASICs. In other words, more memory is required to compute the consensus function. This underlying PoW algorithm is called Ethash. Ethash is an enhanced version of the Dagger [15] and Hashimoto [16] hashes that reduces their computational overhead. During the process of mining, the algorithm pseudo-randomly selects elements from the large shared data set in the blockchain, a directed acyclic graph (DAG). The mining node needs to find a nonce that, together with transaction and receipt data, generates a hash below 2,256 divided by a dynamic target difficulty. If the mining node succeeds in finding

such a nonce, the node has successfully presented a valid proof of work and the block is successfully mined [15]. In effect, Ethereum managed to create an ASIC-resistant, non-outsourceable PoW algorithm by making the main limiting factor not CPU power, but memory [15].

While this consensus algorithm provides substantial advantages over most conventional PoW algorithms, PoS algorithms are believed to outperform their PoW counterparts in the long-term, particularly because of their efficiency properties. This is why Ethereum are planning on transitioning to the hybrid Casper protocol [17] as part of their Ethereum 2.0 roadmap, which is expected to launch in 2020. Casper is a Byzantine Failure Tolerant PoS consensus algorithm, where malicious, or byzantine, nodes are punished by losing their stake, aiming to improve upon traditional PoS protocols. Additionally, Casper provides enhanced transaction finality, which refers to the reverting of transactions and blocks after completion.

2.4.2 Solidity & EVM

Solidity is the high-level, object-oriented programming language used to program smart contracts in Ethereum. It is syntactically similar to JavaScript with contracts behaving like classes. Any Solidity source code must start with a line of the syntax "pragma solidity *version*;", specifying the version of Solidity to be used. Solidity is statically-typed, meaning that types must be specified when developing smart contracts. Solidity types include *bool*, unsigned integers ranging from *uint8* to *uint256*, signed integers ranging from *int8* to *int256*, *address*, *string*, *byte[]* and mappings from some key type to some value type.

```
pragma solidity ^0.5.0;

contract Wallet {

     address payable owner;
     string walletName;

     // constructor to set the wallet owner address
     constructor(string memory walletNameIn) public {
          owner = msg.sender;
          walletName = walletNameIn;
     }

     modifier onlyOwner {
          require (
               // only the owner can call this function
               msg.sender == owner
          );
```

```
19            _;
20      }
21
22      // if owner calls this function then it is executed
23      // otherwise an exception is thrown
24      function withdraw(uint amount) public onlyOwner returns(bool) {
25
26            // checking if the balance is greater than 0
27            // and greater than or equal to the amount
28            if (address(this).balance > 0 && address(this).balance >=
    amount)
29                  owner.transfer(amount);
30                  return true;
31            return false;
32      }
33
34      // getter function to return the name of the wallet
35      function getName() public view returns(string memory) {
36            return walletName;
37      }
38  }
```

Listing 2.2: A Simple Contract in Solidity

Listing 2.2 above shows a simple smart contract wallet implementation in Solidity that stores the wallet's owner address, as well as the name given to the wallet. The contract also includes a function called *withdraw* that transfers a specified amount of Ether to the owner address, as well as a function called *getName* to retrieve the name given to the wallet. An important concept of the EVM is that there are three different types of space: the EVM stack, EVM memory and the EVM long-term storage, in which state variables, like *owner* in the code snippet above, reside in. Only the last storage type does not reset after computations terminate. This is important, because in effect, state variables in an Ethereum contract make the execution of the contract expensive and hence, users will have to pay higher fees. Therefore, contract developers should minimise the use of Solidity state variables in order to optimise the cost of smart contracts.

Functions and state variables in Solidity are public by default, however, one can specify visibility explicitly using one of the following keywords: *external*, *public*, *internal*, *private*. External functions or state variables can be called from other contracts and transactions, however, they cannot be called from within the contract itself (but this.*functionname*() works). Public functions can be called internally or via messages and internal functions are only accessible internally without using the *this* keyword. Lastly, functions declared as private are only accessible by the contract itself and not from derived contracts [18]. Functions declared with the *view* keyword, such as getter functions, do not modify the state of the blockchain, whereas functions declared as

pure promise not to read from or modify the state [19]. Solidity also offers something called function modifiers, which can be used to change the behaviour of functions. For instance in the Solidity code snippet above, the function *withdraw* in line 18 checks whether the caller of the function corresponds to the address stored as the value of the *owner* variable prior to executing the body of the function. If so, the EVM proceeds to execute the function body, otherwise an exception is thrown. Function modifiers can be overridden by derived contracts [20].

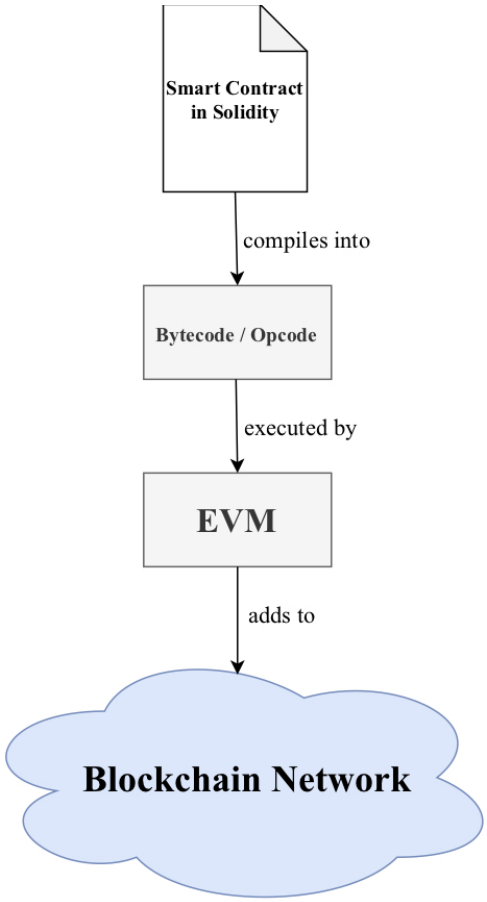


Figure 2.5: EVM Smart Contract Execution

Diagram 2.5 above illustrates how the EVM executes Solidity code. First of all, the Solidty code needs to be compiled into its equivalent low-level, stack-based EVM bytecode. This bytecode encodes a list of bytes, each representing a code for some operation (opcode), which the EVM is capable of executing. Each opcode is used to execute specific tasks. The beginning of the compiled bytecode for the Solidity contract from Listing 2.2 is presented below.

6080604052348015610010576000 80fd5b506040516104a53803806104a5833981018060405
260208110156100335760008 0fd5b8101908080516401000000008111156100 4b576000 80...

The EVM separates this bytecode into its bytes, one byte being represented as two hexadecimal characters. Some opcodes, such as PUSH1, include push data and hence need to be treated differently (by appending additional hexadecimal characters to the instruction). The first few bytes of the above bytecode can be split into the following set of instructions: 0x6080, 0x6040, 0x52, 0x34, et cetera [13]. The first byte, namely 0x60, corresponds to the PUSH1 opcode, which implies that 1 byte of data is pushed onto the stack. Therefore, the full first instruction corresponds to 0x6080. The same logic applies for the next two bytes of the code. 0x52 corresponds to the MSTORE opcode, which writes a *(u)int256* to memory. 0x34 corresponds to the CALLVALUE opcode, which returns *msg.value*, that is the message's funds in Wei, and so on. Finally, after having successfully executed every opcode in the contract, the transaction is pushed to the Ethereum blockchain network.

2.4.3 Security Issues

Smart contracts written in Solidity have suffered from many exploits in the past — the two most prominent being the Parity multi-sig wallet bug and the Decentralized Autonomous Organization (DAO) bug.

In July 2017, the muti-sig wallet attack occurred when a hacker managed to attack three multi-signature wallets containing more than $31,000,000 worth of Ether. The only way this attack could be stopped was by getting a group of white hat hackers to exploit the same vulnerability and hack the remaining wallets. This bug occurred because of a programmer-introduced bug in Solidity code permitted by the Solidity programming toolchain. This is just one example of how Solidity's vulnerability of insecure coding can result in dire consequences. With smart contracts in particular, programmer-introduced bugs can have much more serious consequences as compared to usual coding bugs — if a smart contract on the blockchain contains a small coding error, this cannot be reverted and can lead to cryptocurrency being frozen or lost entirely in the worst case.

```
// User Contract
function moveBalance() {
     wallet.withdraw();
}
...
```

```
6 function () payable {
7       wallet.withdraw();
8 }
```

```
1 // Wallet
2 uint balance = 10;
3
4 function withdraw() {
5       if (balance > 0)
6             // calls default payable function
7             msg.sender.call.value(balance)();
8       balance = 0;
9 }
```

Listing 2.3: DAO Bug Vulnerability in Solidity Code

The DAO bug is a reentrancy vulnerability [21] that took advantage of the fact that Ethereum contracts can call other contracts, allowing for looping through recursion. This attack recursively called Solidity's default *call.value()* function. Because the sender's balance was not updated before the Ether was sent, this allowed the attacker to repeatedly withdraw from a wallet using a fallback function. Listing 2.3 shows Solidity code vulnerable to this attack, where the two *wallet.withdraw()* calls in line 3 and 7 of the user contract loop until the contract's balance becomes empty. This vulnerability enabled the attacker to steal 3.6 million Ether — worth around $70 million.

Further Solidity language vulnerabilities include overflow, underflow, replay attacks and transaction reordering. Generally speaking, Solidity offers a large amount of functions to developers and in effect, the less functionality a programming language offers, the fewer attack vectors there are for hackers to exploit. In other words, the more code goes into a smart contract, the more vulnerable it becomes to security threats. Chapter 8 draws comparison between smart contract code written in Nexus and the code required to implement an equivalent smart contract in the Solidity language.

Moreover, it can be argued that the vulnerabilities mentioned may be caused by the underlying EVM environment, which allows these bugs to occur, by for instance relying on large 256-bit registers, which do not match the low-level instructions of a real machine. This means that even the simplest set of calculations needs to be transformed to be usable for the EVM, which turns out to be an extensive and costly process. The EVM is known to be a low performance virtual machine, making calls to external smart contracts expensive and slow, thus creating another security issue. This is one of the reasons why Ethereum are planning on transitioning to eWASM, Ethereum flavoured WebAssembly, in place of the conventional EVM in the near future (see section 6.1.4).

Lastly, there exists no central source of documentation about known Ethereum security vulnerabilities, but instead these are documented separately and widely spread in

several sources, further contributing to the occurrence of aforementioned attacks [22]. Section 6.1 will elaborate on why certain technologies were used for our implementation, regarding the security, as well as the usability and performance requirements of this project.

2.4.4 Scalability Issues

One of the reasons why existing cryptocurrencies have not been adopted globally as of yet, is the fact that common digital currencies like Bitcoin and Ethereum do not scale well enough to support billions of users worldwide. This is due to their decentralised nature and the fact that transactions need to be processed and validated by every single node in the entire network, heavily limiting scalability. Bitcoin can only process about 4.6 transactions per second (tps), Ethereum can handle about 15 tps, whereas Visa can process up to 47,000 tps, averaging about 1,736 tps [23, 24]. These low numbers present for Bitcoin and Ethereum are due to their hard-coded block size limit and their average block creation time. Scalability in Ethereum is worsened further by the fact that Ethereum provides many different applications running on top of the blockchain, leading to accelerated growth in the number of network nodes that need to process transactions. Additionally, Ethereum permanently stores state variables in the blockchain, making storage extremely expensive and thus further limiting the blockchain's scalability. Furthermore, the Ethereum protocol relies on a generalised state transition function, which makes it unsuitable for machines to apply optimisations such as partitioning, parallelisation of transactions, or the divide-and-conquer strategy [13]. Several measures have been implemented to enhance the scalability of blockchain technology, ranging from alternatives to the proof of work consensus, to blockchain sharding [25], to off-chain payment solutions [26].

To conclude, there is a trade-off between decentralisation and scalability of blockchains, and in order to achieve a more scalable solution, blockchains must either rely on a more centralised system, or make use of other alternatives.

2.5 Libra

Earlier this year, tech giant Facebook have announced their Libra project and published the Libra Blockchain whitepaper [27]. Libra is a permissioned blockchain digital currency to be launched in 2020. When Libra was initially announced, the plan of this project was to implement Nexus on Ethereum as well as the Libra blockchain as its permissioned counterpart. However, because Libra is currently still in its early development stages, it has been proven to be of little use for this project for several reasons. Although we have been in contact with the founders of the Libra Association regarding our needs, we have been unsuccessful in getting the required features for this project in time. Certainly, this is one of the future goals of this project, which will be further discussed in section 9.2.

Libra is reserve-backed by a so-called basket of government bonds and short-term government securities, as well as historically stable currencies, such as the pound, in order to eliminate volatility by giving the Libra Association the ability to adjust the balance of Libra's composition. This effectively stabilises the value of the Libra coin and this is the reason why the Libra coin is referred to as a stablecoin. However, Libra coin is not a stablecoin by definition, as its value is not pegged to a single currency, but instead to four fiat currencies: the pound, the swiss franc, the euro and the yen. Generally speaking, ordinary cryptocurrencies like Bitcoin fluctuate regularly and strongly, which introduces a degree of risk and often makes it undesirable for users to hold or use cryptocurrencies in the first place.

The Libra wallet will be available over WhatsApp and the Facebook Messenger app. Effectively, Libra is trying to make transferring money from your smartphone as simple as it is to send a picture nowadays, "no matter who you are, where you live, what you do or how much you have" [28]. This should be enabled without incurring any fees for exchange rates or international transactions — nowadays international transaction fees average about 7% [28]. Although gas fees remain, just like in Ethereum, these can be regarded as negligible. Using Libra as payment system instead of relying on conventional systems brings the benefits mentioned previously in section 2.1. The main issue the Libra Association is trying to address is the fact that many people do not have access to any financial services and or do not have bank accounts. There are over a billion people who possess a smartphone, but do not have a bank account [28]. Libra is seeking to fill this gap by reducing these barriers to entry and improving access to financial services in order to provide "a new global currency".

2.5.1 Adoption Requirements

As mentioned previously, the most prominent blockchains Bitcoin and Ethereum suffer from scalability issues due to their decentralised nature and the fact that transactions need to be validated by every single node in the entire network. In Bitcoin, users have to wait on average 10 minutes for their transaction to be confirmed and included in a block, which is impractical for a global payment system application that needs to scale to the extent Visa does. In addition to this, when making a transaction in Bitcoin from a single address, all coins located at the address need to be included in that transaction in order to prevent double spending. This means that if an individual owning a single account has 100 bitcoins in their account, but only wants to spend 1 bitcoin in a specific transaction, they will need to wait until the transaction has been validated by the network in order to be able to spend the leftover change, id est 99 bitcoins. In the blockchain world, there has proven to be a trade-off between decentralisation and performance, which is why the Libra blockchain is currently still a centralised scheme so as to provide sufficient levels of scalability. When Libra is launched in 2020 it is expected to have a transaction speed of 1,000 tps, enabled by a proof of stake consensus algorithm, which leverages distributed computing principles and Byzantine fault tolerance [29] (see section 2.5.2). Until Libra is officially launched, the scientific community behind the non-profit organisation is making continuous progress in the research area of scaling consensus technology.

"It must be centralized enough to prevent illicit activity by freezing funds, but decentralized enough not to discriminate against participants based on the use of the funds" [30], as Marco Santori, president and chief legal officer of Blockchain.com, stated in his Twitter post. This is an interesting statement referring to Libra having to provide sufficient levels of privacy while maintaining its power to resolve and deal with unexpected scenarios. Users should be able to make payments without having to prove their identity. A bank account or Know Your Customer (KYC) check should no longer be required in order for users to make a payment, thus resembling the use of cash. At the same time, users should not be identifiable from a set of transactions. In Bitcoin's public chain of blocks, it has been proven to be considerably easy to identify individuals from user behaviour. Monaco et al. make use of transaction features such as timestamps, coin-flows, connectivity, as well as hardware and network latency, to successfully identify Bitcoin users [31].

2.5.2 The Libra Blockchain

The Libra blockchain consists of a distributed database, or ledger, that contains a record of all previous transactions in the network. This record is maintained by the Libra protocol. An open-source prototype of the Libra blockchain has been implemented

in Rust, namely Libra Core [32], allowing developers to validate the Libra protocol and provide feedback to improve the system. Rust was chosen as the implementation language because of its performance, its facilitation of safe coding practices, as well as its support for systems programming [33].

In the Libra network, a set of validator nodes, which currently consists of the 28 Libra Association members, work collaboratively in order to validate transactions and maintain the current state of the ledger S_i [33]. By state, Libra refers to a current snapshot of the data in the blockchain, in other words the state of all Libra accounts on the chain [34]. The data structure behind the global blockchain state is a key-value pair that maps Libra account addresses to their corresponding account values. Unlike in ordinary blockchains, there is no actual chain of blocks present, but instead Libra relies on a database with transaction histories and Merkle trees to ensure immutability. The first state in the blockchain, S_0, is referred to as the genesis state, which defines the key components of the ledger, that being modules like accounts and coins. The first transaction in the blockchain, tx_0, is a special transaction that defines theses modules and resources. Every new transaction, tx_i, changes the state, identical to the state transitions represented in figure 2.4. This makes the Libra ledger a versioned database, where the version number represents the number of transactions, i, processed on the blockchain.

2.5.2.1 PoS Consensus

The Libra blockchain relies on a single consensus algorithm called LibraBFT. LibraBFT is based on the HotStuff framework [35], which is a robust and efficient state machine replication system [29]. It uses a PoS consensus algorithm with Byzantine Fault Tolerance (BFT). Since Libra is not fully decentralised as of yet, the probability of validating and creating a new block in the chain is not proportional to the relative share of the currency the node owns.

"Byzantine faults are worst-case errors where validators collude and behave maliciously to try to sabotage system behavior" [33]. BFT originates from the Byzantine Generals' Problem [36], where a group of generals are spread around an enemy city and struggle to agree on a common decision of attack. In the Libra blockchain, each general represents a validator node in the network and these nodes need to reach consensus on the current system state. Each general gets one vote and the generals exchange their votes via messages. However, messages may be delayed, destroyed or manipulated on the way. Lamport et al. [36] show that this problem is solvable if and only if more than two thirds of the generals are honest nodes. A BFT system therefore implies that even if there exists a failure caused by the Byzantine Generals Problem, such as a node crashing or acting maliciously, the system is resistant and remains active. The following figure illustrates Libra's PoS algorithm with BFT.

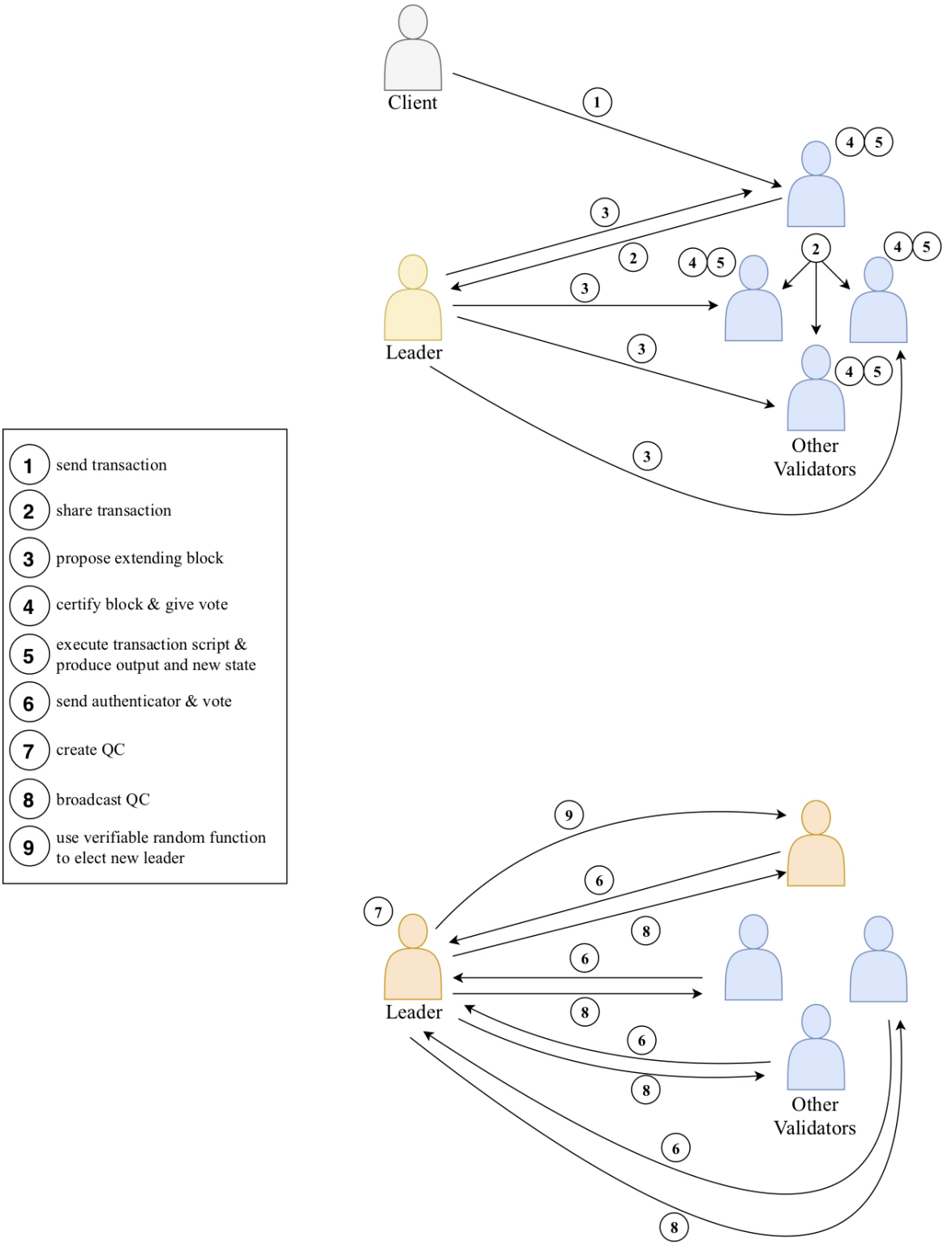


Figure 2.6: The Libra Consensus Mechanism

LibraBFT operates in rounds. Every round k the proposer of the block that was last committed to the chain uses a verifiable random function [37] to determine a new leader. This leader election mechanism is unpredictable and makes Denial of Service (DoS) attacks infeasible, because attackers only get a very small time window to perform such attack. Clients submit transactions to validator nodes (①) who propagate these amongst each other (②). The leader of the current round proposes a block of transactions that correctly extends the currently validated sequence of blocks containing all previous transactions and forwards this to the other validator nodes (③). When validators receive the proposed block they either certify the block or they do not. When certifying the block (④), the Libra VM "speculatively" executes all transactions included in the block by verifying (checking resource, type and memory safety) and then running their bytecode. "Speculatively" implies that the VM executes transactions prior to them being agreed on – because there are no external effects and transactions are deterministic, doing so is safe [33]. When executing transactions, the VM ensures that resources are not lost, duplicated or included in unauthorised transactions. When the VM executes a transaction on the current state of the blockchain, an execution output (consisting of a status code, event list and gas usage), as well as new state are produced. An advantage of this execution protocol is that the output is fully determined by the transaction and the current blockchain state, since execution is deterministic and hermetic [33]. This allows nodes to reach consensus on the same sequence of transactions although the transactions included are executed independently by validator nodes. This also enables the blockchain to re-execute the ledger from the genesis state if needed.

After having certified and executed the transactions included in the proposed block, the new transactions are appended to the current ledger history and a so-called authenticator for the ledger is created. The validator node sends this authenticator together with their signed vote to the leader node (⑥), who collects all votes and creates a so-called quorum certificate (QC) (⑦). In order to reach consensus in this mechanism, the leader needs to prove to the other validating nodes that it has received at least *2f+1* votes for the proposed block, when at most f validators are run by evil nodes. If this number of votes has been received, the leader broadcasts its QC (⑧). The proposed block is committed to the chain if it has been confirmed by two blocks and their corresponding QCs in rounds *k+1* and *k+2*. This algorithm guarantees that all non-malicious validator nodes will eventually commit the proposed block [33]. After the block has been committed to the chain, the current leader uses a verifiable random function to elect a new leader for the next round (⑨).

Clients are also able to audit any validator node and create a replica of the ledger state for any version i to ensure validators execute transactions correctly, creating transparency and accountability in the system [33]. Gas fees in the Libra blockchain are only used to prevent DoS attacks. As usual, transactions with higher gas prices are preferred by validators, whereas ones with lower gas prices may be dropped. This

reduces demand under high load. Similarly, a maximum gas amount is specified to prevent expensive code from being executed. LibraBFT assumes that *3f+1* votes are propagated through the network and guarantees resistance to double-spend attacks as long as at most f validators are run by malicious validator nodes. When validator nodes crash or restart LibraBFT also remains safe, even in the worst case scenario when all validators have to restart simultaneously.

2.5.3 Move

Move [38] is used for programming custom Libra transactions and writing Libra smart contracts. The Move source code is a layer on top of Move bytecode and it can be used to test the Move bytecode verifier and the Move VM. The actual Move source language is currently being developed, however, Move IR has been released as open-source software allowing users to play around with the Libra testnet, test the semantics of Move code and add additional features that can help make the language more usable, efficient or even more secure. Move IR is higher level than assembly code, enabling developers to write human readable code, yet low level enough to convert developed code into Move bytecode usable for the Libra protocol. The final Move source language will be higher level than the current intermediate representation.

Move uses Rust as specification language for the Libra protocol and also steals some design features from the Rust language. For instance, Move makes use of memory ownership and borrowing the same way Rust does. There are two types of Move programs: modules and transaction scripts.

```
module LibraCoin {

  // A resource representing the Libra coin
  resource T {
    // The value of the coin. May be zero
    value: u64,
  }

  resource MintCapability { ... }

  public borrow_sender_mint_capability(): &R#Self.MintCapability {
    ... }

  public mint(value: u64, capability: &R#Self.MintCapability): R#Self
    .T { ... }

  // This procedure is private and thus can only be called by the VM
    internally.
  grant_mint_capability() { ... }

  public zero(): R#Self.T { ... }
```

```
19
20    public value(coin_ref: &R#Self.T): u64 { ... }
21
22    public split(coin: R#Self.T, amount: u64): R#Self.T * R#Self.T {
        ... }
23
24    public withdraw(coin_ref: &mut R#Self.T, amount: u64): R#Self.T {
        ... }
25
26    public join(coin1: R#Self.T, coin2: R#Self.T): R#Self.T { ... }
27
28    public deposit(coin_ref: &mut R#Self.T, check: R#Self.T) { ... }
29
30    public destroy_zero(coin: R#Self.T) { ... }
31
32    public TODO_REMOVE_burn_gas_fee(coin: R#Self.T) { ... }
33 }
```

Listing 2.4: Move Module Sample [39]

The Move language allows developers to publish datatypes via modules, similar to smart contracts in Ethereum. Modules are defined by the address where they were created and are used to declare structs and procedures that allow developers to specify rules for creating, destroying and modifying its declared structs. Structs must be labelled either as resource or unrestricted (also referred to as non-resource). Modules can execute procedures from other modules as well as use their types. However, for security reasons some operations, such as destroying a resource, can only be executed from within the module defining the resource. As of now, Move modules are immutable and once published to the blockchain they cannot be modified or destroyed, unless a hard fork is carried out. The Libra Association is currently looking for ways to allow safe updates of published modules. An Ethereum smart contract is published under a single account address, whereas a module in Move can define various resources published at different addresses.

Move allows developers to create custom resources, such as the Libra coin. This is a different concept from the one used by common blockchains like Bitcoin or Ethereum, where the Bitcoin and Ether currencies are of special type. Move resources follow the Rust ownership model and abide by the rules defined in their module. Resources contain the name of the resource type, the module that defines the resource as well as the address where the module is stored. This is illustrated in figure 2.7 above, where address 0x1 created one module named *MyModule1* as well as two resources, where one of them is of the T type that was created in the *LibraCoin* module at address 0x0. To prevent double spending and reentrancy attacks, a Move resource can never be copied. It can only be moved or borrowed between storage locations. While the EVM allows reentrancy attacks, Move does not.

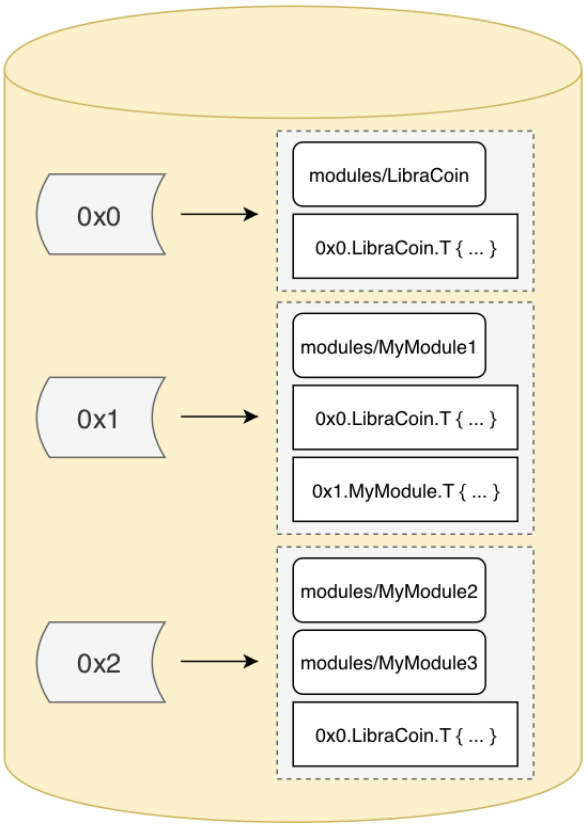


Figure 2.7: Example Global State of the Libra Blockchain [38]

Non-resource types are unrestricted, because the copy and destruction rules previously mentioned are not enforced for values of these types. Unrestricted values, such as *address* or *u64* can be copied or moved, whereas values tagged as a resource can only ever be moved – the bytecode verification will trigger an error otherwise [38]. Furthermore, resource destruction can only be carried out by the module that initially created the resource. For instance, only procedures of the *LibraCoin* module are able to create or destroy values of the *LibraCoin* type, making types in the Move language transparent only within its declaring module. These safety guarantees for resources are imposed by the Move VM and allow for safe developing of Move code. This also creates a strong degree of data abstraction in Move.

This structure makes Move programming similar to object-oriented programming, which deals with classes, objects and methods. However, in Move a module may declare multiple or no structs at all. There is also no such thing as public fields, implying that no data can be accessed outside of its declaring module. Additionally, all procedures in a module are static and thus 'this' or 'self' are non-existent within a procedure. As of now, developers are unable to publish custom modules that declare

their own resource types, however, this is something that the Libra Association is going to make available before the launch of the Libra network.

```
//! no-execute

// Simple peer-peer payment example.

// Use LibraAccount module published at account address
// 0x0...0 (with 64 zeroes). 0x0 is shorthand that the IR pads out to
// 256 bits (64 digits) by adding leading zeroes.
import 0x0.LibraAccount;
import 0x0.LibraCoin;
main(payee: address, amount: u64) {
  // The bytecode (and consequently, the IR) has typed locals.
  // The scope of each local is the entire procedure. All local
  // variable declarations must be at the beginning of the procedure.
  // Declaration and initialization of variables are separate
  // operations, but the bytecode verifier will prevent any attempt
  // to use an uninitialized variable.
  let coin: R#LibraCoin.T;
  // The R# part of the type above is one of two *kind annotation*
  // R# and V# (shorthand for "Resource" and "unrestricted Value").
  // These annotations must match the kind of the type declaration
  // (e.g., does the LibraCoin module declare `resource T` or
  // `struct T`?).

  // Acquire a LibraCoin.T resource with value `amount` from the
  // sender's account.  This will fail if the sender's balance
  // is less than `amount`.
  coin = LibraAccount.withdraw_from_sender(copy(amount));
  // Move the LibraCoin.T resource into the account of `payee`.
  // If there is no account at the address `payee`,
  // this step will fail
  LibraAccount.deposit(copy(payee), move(coin));

  // Every procedure must end in a `return`. The IR compiler is
  // very literal: it directly translates the source it is given.
  // It will not do fancy things like inserting missing `return`s.
  return;
}
```

Listing 2.5: Move Transaction Script Sample [39]

A transaction is an authenticated wrapper around a Move bytecode program and every transaction includes a transaction script (along with the sender address, sender public key, gas price, maximum gas amount and sequence number) [33]. Transaction scripts are declared with the *main* keyword and invoke procedures of some module, thus updating the global blockchain state (see figure 2.7). The transaction script above is a simple script that transfers a given *amount* of Libra coins to the specified *payee*

address. Line 9 implies that the script is using the *LibraCoin* module located and created at address 0x0. Note how in this script the *amount* value and the *payee* value are copied, but the *coin* value gets moved. This is because *coin* is a resource struct of type *T*, which changes ownership and hence needs to change storage location from one account address to another, whereas the *amount* and *payee* values are kept for future use. Since the latter two values are unrestricted, they could have also been moved in this case. If *coin* was not moved, for example if the code in line 31 was commented out, the VM would throw an error when verifying this transaction script, preventing Move programmers from "losing track of a resource". Similarly, after moving the *coin* value in line 31 this value becomes inaccessible. "These guarantees go beyond what is possible for physical assets like paper currency" [38]. The safety guarantees that the Libra type system enforces, including resource safety, memory safety and type safety, allow the Libra Association to implement a secure fungible asset: the Libra coin.

2.5.4 Scalability

Generally speaking, just like with any other consensus algorithm, the LibraBFT protocol performance is dependent on how efficiently the validator nodes controlling the network interact with each other. As a result, the block commit latency is solely restricted by the network latency present between validator nodes [33]. LibraBFT chose to extend HotStuff over other BFT-based protocols, particularly because of its promising performance. By extending the original HotStuff protocol, LibraBFT provides enhanced safety, liveness and optimistic responsiveness. Just like other cryptocurrencies, LibraBFT makes use of Merkle trees for transaction verification and authentication. However, LibraBFT relies on a single Merkle tree for the entire ledger history instead of relying on one Merkle tree for each block in the chain, like Bitcoin does. In Bitcoin, this setup turns out to be very inefficient when having to look up a specific transaction in a previous block, because every subsequent transaction needs to be fetched and processed. To prevent this, Libra uses a Merkle tree accumulator to form Merkle trees. This makes appending to the tree efficient (O(log n) storage used) and verifying that one Merkle tree is the composition of two others can also be done efficiently [40]. Additionally, LibraBFT makes use of sparse Merkle trees to shard the ledger history among numerous machines, allowing the processing of updates and initial validations to be done in parallel. Sparse Merkle trees also expand capacity. As described in the previous subsection, the LibraBFT protocol reaches consensus in three network communication rounds and is not suffering from any delay introduced by the proposing and voting steps. Although LibraBFT is still a prototype implementation, it is expected to have a 10-second finality time between a client submitting their transaction and the transaction being included in a block committed to the chain [33]. This is made possible, because PoS algorithms based on BFT generally require consensus before the next block is formed, which permits settlement finality prior to

block generation as well as a speedup in transaction times [41]. The Libra Association also confirmed that many payments will occur off-chain, for instance through payment channels. The aforementioned reasons suggest that 1,000 tps are indeed achievable for the Libra protocol. Faster transaction times and increased power as well as energy efficiency lead to enhanced scalability by this solution as opposed Ethereum's PoW consensus mechanism.

2.5.5 Future Impact

Libra is not completely decentralised nor fully permissionless yet. It is currently controlled by its 28 Libra Association members, including companies like Uber and PayPal, in order to meet its reliability and scalability requirements. Each of these members gets a vote in the decision-making process of reaching consensus, and together they can decide on a conclusion. Libra is aiming to become fully permissionless in the near future [42] and meet the user needs and requirements referred to in previous subsections. Right now, there is an open-source prototype implementation of the blockchain available, turning Libra's development process into "a global collaborative effort to advance this new ecosystem" [33].

The impact Libra will have on the blockchain world as well as our society, and whether Libra will reach mainstream adoption, is unknown as of yet. While the potential of Libra to revolutionise our modern payment system is immense, its impact and success are strongly dependent on whether people will adopt this cryptocurrency. There are many people who simply prefer using cash as opposed to using their credit card, as they do not want their credit card supplier to know what exactly they spend their money on. In the ordinary blockchain, every single transaction is public. This is one of the topics that puts people off using blockchain. Furthermore, Calibra being a subsidiary of Facebook, there have been critics questioning whether it is a good idea to trust Libra with their money, considering Facebook's bad reputation of customer privacy violations in the past, such as the Cambridge Analytica data leak scandal in 2018, where millions of Facebook profiles were harvested [43]. These are factors that will strongly affect the adoption of Libra in the long-term and Libra will need to provide scalability, privacy and fungibility of its currency in order to persuade the public. Libra will also need to remain compliant with existing domestic and international regulations while offering the features promised. As Libra co-creator David Marcus stated during the Facebook Libra antitrust hearing on July 16th: "Facebook will not offer the Libra digital currency until we have fully addressed the regulatory concerns and received appropriate approvals" [44].

2.6 Combinators

We define a combinator to be a mathematical function that acts as a building block for our language. Combinators allow us to construct complex compositional smart contracts out of simpler ones, define and process contracts, as well as to evaluate them to find their respective values (in terms of some cryptocurrency). The concept of combinators is very useful in financial engineering, as the financial industry is surrounded by a great deal of technical jargon and a specific set of technical contracts such as European Options and American Options are commonly referred to and reused. It would be inefficient to define these separately. Instead, we will rely on specific low-level combinators to define such financial contracts, abstracting away from the underlying complexities of financial contracts while allowing users to concentrate on the logic of Nexus contracts. Using a set of carefully-defined combinators, also referred to as a combinator library, makes it easier to describe new contracts and reduces the cost of analysing and performing computations over new contracts. New combinators can be easily added from already defined ones and separating combinators simplifies the semantics and enriches the algebra of contracts [1]. Peyton Jones et al. use Haskell as implementation language to define their domain-specific combinator library. This is mainly due to Haskell's functional programming nature and the language's lazy evaluation feature, which are well suited language elements for this specific domain. In their paper, it is concluded that a great number of financial contracts can be defined from a small set combinators, which this paper is going to build upon by implementing a domain-specific combinator library in JavaScript (see section 5.4).

Chapter 3

Related Work

The first commercialised blockchain was Bitcoin introduced by Nakamoto [4]. Following the success of this paper, the demand and value of Bitcoin rose drastically, effectively popularising blockchains. Although Szabo first introduced the idea of leveraging decentralised ledgers for self-executing agreements enforced not by law, but by hardware or software in 1994 [6], coining the term 'smart contract', the idea was not implemented until after the Bitcoin blockchain was launched. Nowadays, the original meaning of the smart contract concept has gotten a little lost and the blockchain industry uses the term 'smart contract' to refer to any kind of transactional computer program on the blockchain. When Buterin launched Ethereum [12] in 2014, smart contracts quickly gained popularity. Initially, Ethereum smart contracts could only be developed in Solidity, Ethereum's high-level programming language targeting the EVM. Numerous attacks occurred on the Ethereum blockchain and huge amounts of Ether were locked and stolen, leading to the realisation of Ethereum's underlying security vulnerability: Solidity. As a result, in the past couple of years a lot of research has gone into writing safer smart contracts.

First of all, efforts have been made to teach smart contract developers to write Solidity code in a more secure way in order to prevent potential security pitfalls [45]. Furthermore, SolidityX [46] was invented by an organisation called Loom Network. This represents an example of a programming language that is 100% compatible with Solidity, while being designed to provide only methods that are secure against known attacks and commonly produced coding bugs. Moreover, WebAssembly (Wasm) was invented to allow developers to compile an extensive set of high-level programming languages [47], including prominent languages like C, C++, Java and Rust, into Wasm code, which can be further compiled into EVM bytecode to be executed on the Ethereum blockchain.

Peyton Jones et al. [1] convey the idea of using a combinator library in a functional programming language in order to give a formal description to financial contracts that allows for efficient composing of complex financial contracts out of simpler ones. By using a declarative language, the authors allow their implementation to be easily portable to different environments and implementable in any programming language. This paper revolutionises the way contracts may be described and evaluated. However, this has only been put into practice in partial Haskell and C++ implementations. Additionally,

the language proposed in the paper is limited and does not implement conditionals, which can be highly valuable for this matter. The authors also do not validate their approach in a real-world environment and financial setting. RISLA [48] is a similar domain-specific language for financial contracts. While this object-oriented language is more stateful than the implementation proposed by Peyton Jones et al., it is less declarative, difficult to extend and the language itself is very broad offering an extensive amount of datatypes and functions. Schrans et al. created a language called Flint [49], a promising statically-typed, contract-oriented programming language targeting the EVM. However, Flint is a general purpose language rather than a domain-specific language, thus, Flint is more vulnerable to buggy financial contracts. Furthermore, Flint lacks ways of simulating and visualising transaction orderings and the language's compiler is targeted at the Ethereum blockchain only.

In addition to the above, technologies have been made available that make use of entirely new blockchains acting as an alternative to existing options. Lisk [50] is an example of such blockchain with a user-friendly interface that does not rely on Solidity at all. Lisk uses JavaScript, making it attractive to a wide audience of developers. However, Lisk is known for having an extended block time, caused by subsequent missed block results producing a 10 seconds delay, resulting in usability and performance issues. As previously mentioned, the Libra Association are launching Libra, a permissioned blockchain that uses Move as its high-level programming language for smart contracts in 2020 [27]. Enterprise blockchain firm Digital Asset have also published their own smart contract programming language called Digital Asset Modelling Language (DAML) [51], specifically designed for companies to synchronise multi-party business processes. DAML is a functional programming language influenced by Haskell with the goal "to become the industry standard smart contract language supported by a wide variety of distributed ledger and database platforms" [52]. DAML distinguishes itself from other smart contract blockchain projects by its ability to allow efficient modelling of very complex business workflows at scale. DAML also enables the details of a contract to be viewable only by authorised parties. DAML has already been integrated with several leading blockchain platforms including the VMWare Blockchain platform, Hyperledger's platform for building, deploying and running distributed ledgers called Sawtooth [53], as well as Amazon's relational cloud storage database service Aurora. DAML seems to be facing a very promising future in the smart contract development world.

Nowadays, there are numerous different programming languages for writing smart contracts, each with their own purpose and advantages and there is no single language dominating this field. This paper proposes a new high-level programming language based on a set of carefully-defined combinators taken from [1]: Nexus. Nexus allows users to write safer financial smart contracts in a declarative way at ease. Nexus will initially be using a minimal smart contract implementation in Rust, which is compiled into EVM code for the Ethereum blockchain via Wasm. However, thanks to our declar-

ative language design and clear separation of blockchain and application layer, Nexus can be ported to any smart contract programming language or blockchain.

Chapter 4

Requirements and Specification

This section discusses the requirements and specification that are needed for the success of this project. It describes which tasks the software must be able to perform and what it should not do. It also outlines the tasks of different system components. The requirements definition of this system consists of functional and non-functional requirements.

4.1 Functional Requirements

The functional requirements help the user, who is assumed to be an individual with little to no blockchain development experience, understand what the system and software are capable of and what actions the user should be able to perform. The following functional requirements are taken into account for the development of the system:

F1 The system shall allow users to easily construct smart contracts via a user-friendly interface of the web app;

F2 The system shall allow user input to be read, parsed, and used to create a smart contract instance;

F3 The system shall allow created contract code to be deployable on a blockchain via some blockchain client;

F4 The system shall allow users to select a contract holder and contract counter party address for the deployment of a contract;

F5 The system shall allow users to select an oracle address that will provide any observable values when a contract is acquired;

F6 The system shall allow users to deposit an arbitrary amount of Ether into both parties' accounts;

F7 The system shall allow users to add their own combinator definitions to extend the existing language;

F8 The system shall visualise composed contracts in a table;

F9 The system shall present users with the option to choose a subcontract whenever a disjunction contract is provided;

F10 The system shall present the user with error logs in case a provided contract is syntactically incorrect;

F11 The system shall allow users to acquire non-expired supercontracts from the table of pending contracts;

F12 The system shall allow users to calculate the value of pending supercontracts for any given time in the future;

F13 The system shall allow users to estimate the gas costs of composed contracts prior to execution;

F14 The system shall allow expired contracts to become unavailable;

4.2 Non-Functional Requirements

The following non-functional requirements are taken into account for the development of the system. 'Real-time' refers to the short period of time between the occurrence of an event and the triggered system response. Real-time responses should be in the order of milliseconds.

NF1 Contract transactions should be processed and sent to the blockchain network in real-time;

NF2 Contract transactions should be confirmed in real-time;

NF3 Expired contracts should become unavailable and labelled as 'expired' in less than 30 seconds after their time of expiry;

4.3 Other

Other requirements that are taken into account for the development of this project include:

- **Usability:** The Nexus end-user will interact with the system via the user interface of our web application. This interface should be simple and clear, so that users can easily operate the software. The grammar of Nexus itself should also be straightforward in order to let non-developers easily create complex custom-defined smart contracts.
- **Compatibility:** This is the capability of the software to be adaptable to different environments with ease. The smart contract code should be minimal, in order to allow implementation of the same functionality in any high-level smart contract programming language. The software should limit the use of external libraries and instead rely on self-implemented code that can be easily adapted to different environments and rewritten in other coding languages. Projects that rely on a vast amount of external libraries often suffer from dependency issues when other dependencies as well as self-implemented code need to be updated because of a library update.
- **Performance:** It is important that the user interface works smoothly and time lags should be prevented. Error messages or alerts should be displayed if an action times out. In the blockchain world performance is an absolute key factor for usability, which is why background compilation and contract deployment should not introduce any time lags.
- **Maintainability & Reusability:** New features and code can be easily implemented and added. Optimisations that add complexity should be avoided unless they provide a considerable benefit. This can be achieved through a clear architectural code structure. Testing throughout the development of the project can also add to this and additionally ensures high code coverage. The goal is to allow average developers to add small modifications to the language in one place, without affecting the functioning of the remaining application, furthering the vision of Nexus as an open-source project.

Chapter 5

Design

This chapter will elaborate on the architectural system design and how different components interact with each other. It will also outline the key components of the system design, including oracles and observables, the use of collateral, as well as the grammar of our language.

5.1 Architectural Design

This section will describe the system architectural design for the development of the project and how different components of the system interact with each other. The goal of the design architecture is to make the front-end as simple as possible for the end user to interact with, while structuring and separating back-end components in order to preserve efficiency, compatibility and reusability of the software. The end user of this system is assumed to be an individual with little to no blockchain development experience.

As illustrated in figure 5.1, a pre-defined Rust contract file storing variables for each contract party address as well as several functions to interact with the contract, such as a transfer function and a function to retrieve balances, is compiled into a Wasm binary file using a Rust tool called Cargo, and further compiled into a JSON contract ABI using the Wasm binary toolkit. This contract ABI code is eventually used for the deployment of the smart contract on the blockchain. The above occurs when the user runs the build script in the root directory. Upon starting the application on a local server, the user is presented with a web portal written in HTML combined with JavaScript and CSS accessible at *http://localhost:9001/*. This web portal allows the user to supply two account addresses for the deployment of a new smart contract instance between two parties. The JavaScript code makes use of this input together with the created contract ABI, passing the two addresses directly to the contract constructor and storing these inside the contract variables for deployment. This will add the smart contract transaction to the current set of unconfirmed transactions for it to be validated and mined by the other nodes in the network. When the block containing the contract deployment transaction has been successfully mined, the contract instance is ready to be used. The user is then able to interact with the specific contract instance

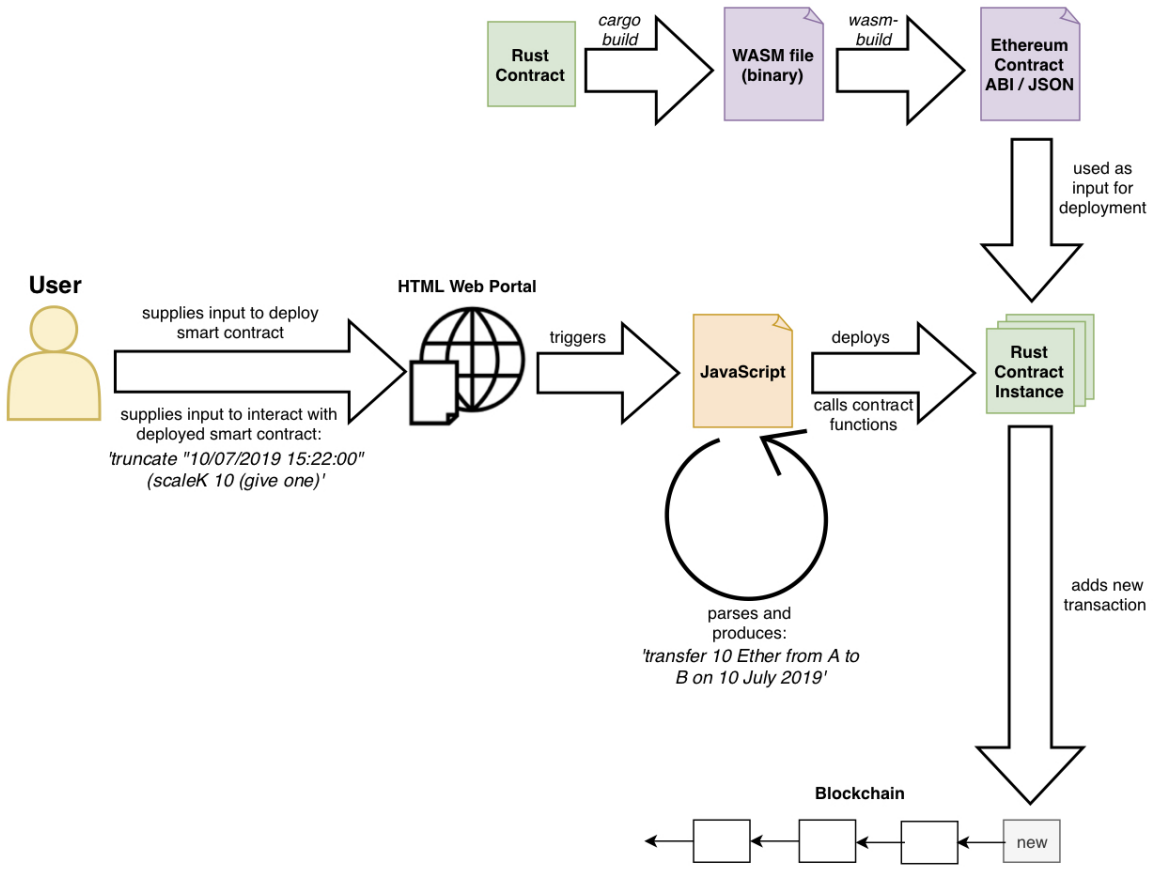


Figure 5.1: System Architecture Design Diagram

by depositing a certain amount of Ether into each account. This deposit acts as an insurance for both parties to receive the funds promised to them and needs this step needs to be done prior to adding any smart contract transactions. This kind of backup is referred to as collateral in the blockchain industry (more about this in section 5.3). Once both accounts have deposited their collateral, their balances stored within the smart contract are initialised accordingly and the two parties can start interacting with each other. The smart contract needs to explicitly store the account addresses and their balances, because any specific smart contract instance can only access values stored within that instance. For example, JavaScript code is able to access the current on-chain Ether balances of each account, however, the smart contract itself cannot. Therefore, the smart contract needs to keep track of the two addresses by storing these together with their respective balances inside contract variables.

In order to add any transactions between the two contract parties, the user must supply a contract (conforming to the language syntax discussed in section 5.4) via the input text field in the web portal. When this input is sent, the JavaScript code will parse

the contract string and produce the corresponding output — a smart contract transfer function call with the supplied input parameters, such as the amount, expiry date, sender and recipient address of the transaction (more on the parsing process in section 6.2). Finally, this transaction is added to the blockchain and executed accordingly upon confirmation.

5.2 Oracles & Observables

Contracts often use quantities measured at a certain date. For example, a party may pay an amount equal to the temperature in London or receive an amount equal to the current 3-month LIBOR spot rate [1]. Naturally, these two amounts are time-varying and hence will be referred to as observables throughout this paper. In order for a contract to successfully execute, it is imperative that both parties involved agree on the value of such an observable, because for example weather.com may provide a different temperature to the one provided by the BBC. This proves the need for an oracle. Prior to being able to set up a contract between two parties, both accounts must agree on a third party account address. This account will act as the oracle for all smart contracts between the two parties, providing the value of any observables. At the time of acquirement of a contract, the oracle provides any external data, such as an observable value, which both parties will agree on. This results in there being no dispute between the parties about any observable values.

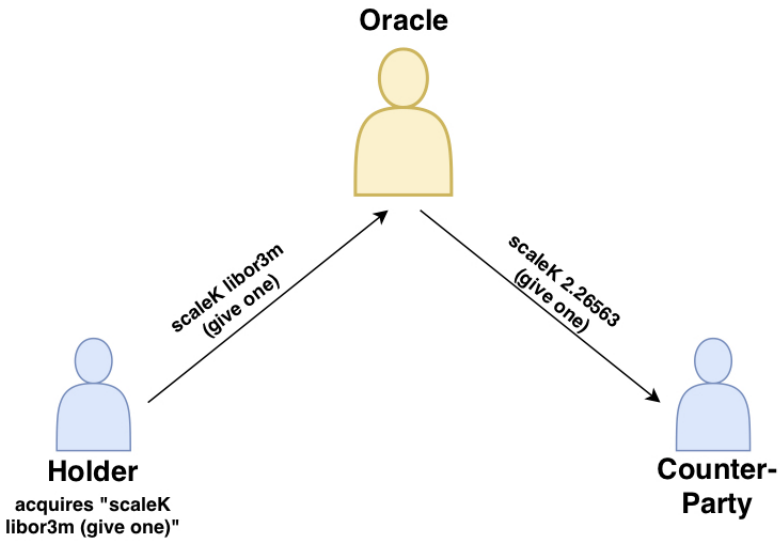


Figure 5.2: Oracle and Observables

The oracle is implemented as a JavaScript object with a method to return the current temperature in London and another method to return the current 3-month LIBOR spot rate. Our application provides 5 pre-defined oracles that return different values for the two observables taken from [1]. Extending this set of observables would be trivial to do. While in this project the values of these observable are pre-defined, this can easily be adjusted so that the oracles instead rely on external API requests that provide the required values. When a contract is acquired any observable contained within the contract is replaced by the corresponding value provided by the agreed upon oracle.

5.3 Collateral

What happens if the terms of a contract cannot be satisfied? For instance, if a contract requires party A to pay party B 10 Ether, however, at the time of acquirement of the contract party A does not have sufficient Ether in their account. How can this case be handled? In this application both parties firstly need to deposit an agreed upon amount of Ether into the smart contract prior to creating transactions. This amount acts as a backup or insurance for both parties to receive the funds promised to them. Such kind of backup is referred to as collateral in the blockchain industry. Once both accounts have deposited their collateral, their balances stored within the smart contract are initialised accordingly and the two parties can start interacting with each other via transactions. Moreover, a user can only add a new contract if the sending account holds enough Ether to execute all pending transactions as well as the newly added transaction at any given time in the future. In other words, the absolute value of the new transaction (negative from the sender's point of view) must be less than the sum of all maximum absolute values of all pending transactions (from the sender's point of view) subtracted from the sender's current balance. For instance, if a user tries to add a new contract c_4 the following equation must be satisfied in order for it to be accepted to the set of pending contracts.

$$max(|V(c_4)|) \geq balance(sender) - (max(|V(c_1)|) + max(|V(c_2)|) + max(|V(c_3)|))$$

$$\text{, where } max(|V(c_i)|) = |V(c_i)|(t_1) \wedge \forall t_2 \neq t_1.|V(c_i)|(t_1) >= |V(c_i)|(t_2) \tag{5.1}$$

If this is not the case, the sending party is asked to top up their collateral. Because contract values are time-varying (see section 6.2.1.2), by the maximum value of a contract we refer to the value of a contract at the time at which it is worth the most. This adds another layer of insurance for both parties to receive the funds promised to them and a contract terms dissatisfaction scenario is effectively being made impossible.

This system design resembles the design of a bi-directional payment channel. A bi-directional payment channel is a 2-party channel blockchain scalability solution, where both parties need to deposit a collateral prior to interacting with each other on the channel, just like in our web application. Each party makes an initial on-chain transaction, in order to deposit an agreed upon amount of Ether stored as collateral. Once both parties have done so, they can interact with each other via multiple off-chain transactions and their off-chain balances update accordingly until these are synchronised with the blockchain. Every new transaction results in a new state of account balances, as illustrated above. This not only results in lower costs for both parties

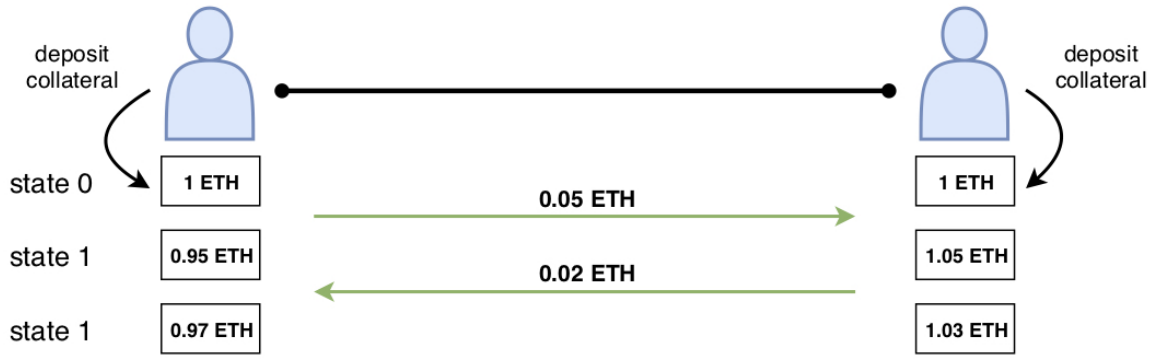


Figure 5.3: Contract Interaction Between Two Parties

making transactions, since on-chain transaction fees are only paid for the initial setup and the final synchronisation step, but also increases contract execution speed, as these off-chain transactions do not need to be propagated and mined by other nodes in the main network. This makes the off-chain transactions virtually invisible for the main network. Security of such design relies on 2-of-2 multisig addresses and hashing. While in our application all transactions happen on-chain, this scalable design is something that should be considered for future versions of the software.

5.4 Grammar

$$
\begin{aligned}
SUPER\_CONT &\rightarrow SUPER\_CONT\ CONJ\ SUPER\_CONT \mid SUB\_CONT \\
CONJ &\rightarrow and \mid or \\
SUB\_CONT &\rightarrow scaleK\ NUM(SUB\_CONT) \mid get(SUB\_CONT) \mid truncate\ DATE(SUB\_CONT) \\
&\quad \mid give(SUB\_CONT) \mid TERM \\
TERM &\rightarrow one \mid zero \\
NUM &\rightarrow ([1-9]\ DIG*(\ .\ DIG*[1-9])?) \mid (0\ (\ .\ DIG*[1-9])?) \\
DIG &\rightarrow [0-9] \\
DATE &\rightarrow ((0?\ [1-9]) \mid ([12]\ DIG) \mid (3\ [01])) \ /\ ((0?\ [1-9]) \mid (1\ [0-2])) \ /\ (DIG\ DIG\ DIG\ DIG) \\
&\quad \backslash\ s\ ((0\ DIG) \mid (1\ DIG) \mid (2\ [0-3])) : ([0-5]\ DIG) : ([0-5]\ DIG)
\end{aligned}
$$

Figure 5.4: Initial Representation of the Language Grammar

The grammar of Nexus should be straightforward in order to let non-developers easily create complex custom-defined smart contracts. Figure 5.4 illustrates the grammar of our language, specifying the syntax of Nexus contracts, thus identifying which contracts are well-formed programs. This entails most of the primitives for defining contracts mentioned in Peyton Jones' paper [1]. Our language consists of multiple combinators, which can be seen as functions that build contracts. These combinators allow us to construct compositional smart contracts out of simpler ones, define and process contracts, as well as to evaluate them to find their respective values in Ether. Every contract c has a contract holder A, who owns and acquires the contract, and a contract counter-party B. One of the simplest contracts between two parties would be: "Receive 1 Ether". In our language this is equivalent to the **one** combinator. This would transfer 1 Ether from the contract counter-party to the contract owner. The combinators **one** and **zero**, which is equivalent to saying "Transfer no Ether" and thus has no impact on the current state of accounts, are the two lowest level combinators in our language. These are used to construct more complicated contracts and every contract includes either **one** or **zero**. Additionally, a contract c has a horizon, which is the latest date at which c may be acquired. By default, a contract's horizon is infinite, meaning that the contract can be acquired at any time and the contract does not expire. If A wanted to pay B 1 Ether, the equivalent contract in Nexus would be **give one**. Technically, this is the opposite of the **one** contract and the **give** combinator can be applied to any contract c in order to make the contract owner pay the contract counter party the value of the contract. Naturally, when a contract c is acquired one party obtains the value of c, while the other party obtains the value of **give c**. Furthermore, **truncate t c** can be used to limit the horizon of c to the smaller of times t and c's original horizon. This implies that c can only be acquired on or before that horizon, because after that time has passed the contract will have expired. The **get c** combinator is

used to acquire a contract at its horizon. For obvious reasons, this combinator can only be used in combination with the **truncate t c** combinator. Figure 5.5 below lists all combinators defined in our language.

Combinator	Meaning
zero	This contract may be acquired at any given time, has an infinite horizon and has no rghts or obligations.
one	This contract may be acquired at any given time, has an infinite horizon and pays the contract holder 1 Ether.
give (c)	This contract pays the contract counter-party the value of c upon acquirement. The rights and obligations of contract c are reversed.
c_1 **and** c_2	This contract executes both underlying subcontracts c_1 and c_2 upon acquirement.
c_1 **or** c_2	This contract gives the contract owner the choice to execute either c_1 or c_2 upon acquirement.
truncate t (c)	This contract is equivalent to contract c with c's horizon trimmed to time t. The final horizon of this contract will be the minimum horizon out of t and c's original horizon.
scaleK k (c)	This contract is equivalent to contract c with c's value of Ether to be transferred being multipled by k.
get (c)	This contract is equivalent to contract c, however, if acquired, c must be acquired at its horizon date. Naturally, this combinator can only be used in combination with the truncate combinator.

Figure 5.5: Combinators for Defining Nexus Contracts [1]

In the web application contract expiry is handled by a timer that constantly monitors the list of pending contracts and checks if any contract within that list has only just expired. If a lower-level contract that does not contain the **get** combinator expires, it is removed from the list of pending contracts and its state in the webportal is updated to be *expired*. If a contract containing the **get** combinator expires, the contract is executed as usual, triggering MetaMask and asking the user to sign the transaction. This check for expiring contracts is automatically executed every minute. If a contract requires

the transfer of an amount different to 1 Ether, the **scaleK k c** combinator can be used. This operator is used to multiply the value of contract c by constant k. For instance, **scaleK 100 (give one)** will transfer 100 Ether to the contract counter party with infinite horizon, whereas **scaleK 100(get(truncate "24/12/2020 23:33:33"(give one)))** will transfer 100 Ether to the contract counter party on the exact date and time of the horizon, that is on 24/12/2020 at 23:33:33.

The context-free grammar in figure 5.4 is presented in the Backus-Naur form (BNF), showing the given production rules that describe all possible combinations of contract strings in our formal language. In each production rule, non-terminal symbols are placed on the left-hand side of the rule, while the right-hand side of each rule lists the possible replacements for each non-terminal. It can be stated that our language is not regular, since there are several rules that have more than one non-terminal in their right-hand side. Our grammar can be used to check whether a contract provided by the user is grammatically and syntactically correct. For instance, the contract string **get(truncate "24/12/2020 23:33:33"(give one)) and zero** will be verified in the following way:

$$
\begin{aligned}
&get\ (truncate\ \text{“}24/12/2020\ 23:33:33\text{”}(give\ (one)))\ and\ zero \\
&\quad = SUPER\_CONT \\
&\quad = SUPER\_CONT\ CONJ\ SUPER\_CONT \\
&\quad = SUB\_CONT\ CONJ\ SUPER\_CONT \\
&\quad = get(SUB\_CONT)\ CONJ\ SUPER\_CONT \\
&\quad = get(truncate\ DATE\ (SUB\_CONT))\ CONJ\ SUPER\_CONT \\
&\quad = get(truncate\ \text{“}24/12/2020\ 23:33:33\text{”}\ (SUB\_CONT))\ CONJ\ SUPER\_CONT \\
&\quad = get(truncate\ \text{“}24/12/2020\ 23:33:33\text{”}\ (give(SUB\_CONT)))\ CONJ\ SUPER\_CONT \\
&\quad = get(truncate\ \text{“}24/12/2020\ 23:33:33\text{”}\ (give(TERM)))\ CONJ\ SUPER\_CONT \\
&\quad = get(truncate\ \text{“}24/12/2020\ 23:33:33\text{”}\ (give(one)))\ CONJ\ SUPER\_CONT \\
&\quad = get(truncate\ \text{“}24/12/2020\ 23:33:33\text{”}\ (give(one)))\ and\ SUPER\_CONT \\
&\quad = get(truncate\ \text{“}24/12/2020\ 23:33:33\text{”}\ (give(one)))\ and\ SUB\_CONT \\
&\quad = get(truncate\ \text{“}24/12/2020\ 23:33:33\text{”}\ (give(one)))\ and\ TERM \\
&\quad = get(truncate\ \text{“}24/12/2020\ 23:33:33\text{”}\ (give(one)))\ and\ zero
\end{aligned}
$$

Figure 5.6: Contract String Verification by Grammar

First of all, the contract string is replaced by the SUPER_CONT non-terminal symbol. At each of the following steps one of the rules from figure 5.4 is applied to one of the non-terminals in the current string (in the example given we are using leftmost derivation, that is we always try out the leftmost option on the right-hand side first). For instance, in line 2 the first rule is applied to replace SUPER_CONT by SUPER_CONT CONJ SUPER_CONT. This process is repeated until we have a string containing no more non-terminals, implying that no replacement rule can be further applied. In this case, it can be confirmed that the initial contract string is grammatically and syntactically correct and valid to be used for the application. If no such string can be found, the contract has been proven to be invalid by our language specification and an error will be raised to notify the user. Figure 5.4 is a simplified presentation, as it is neglecting the use of parenthesis in the scenario in which a single contract is composed of multiple sub contracts. Without the enforcement of such parenthesis our context-free grammar would be ambiguous. In other words, if we have a contract defined as **give zero and one or one**, an ambiguous grammar can parse the contract as **(give zero and one) or one**, but also as **give zero and (one or one)**. Naturally, the meaning behind these two contracts is completely different. Thus, there would be more than one correct parse tree. Nexus handles this problem by parenthesising compositional contracts and thus enforcing explicit blocks.

By relying on a language based on a set of carefully-defined combinators, we further enhance the security of our system, since our combinator functions can be used to prove mathematical properties about our language and thus formally verify the behavioural space of Nexus (see section 6.2.3).

5.5 Combinator Extensibility

Having multiple low-level combinators defined separately, rather than relying on higher-level primitive definitions, enables us to simplify the semantics of our language and enrich the algebra of our contracts [1]. This also allows us to form complex contracts by combining multiple simpler contracts. Additionally, the use of precisely defined combinators enables us to easily extend our language. In the web application the user is able to extend the language by adding any of their own combinators, composed of multiple already existing definitions. For example, the user may add **andGive = and give;**. This will add the new definition to the set of combinators and whenever a user composes a contract containing **andGive**, this will be replaced by **and give**. This process simply looks for the intersection set between the list of keys from the map of user-defined combinators and the words in the contract string and replaces each element in this set by the corresponding value present in the map. This considerably improves usability, as users may reuse the same contract or contract semantics in multiple transactions. A very common example of a contract that may be frequently used is the zero-coupon discount bond: "Receive 100 Ether on 1st January 2020".

Figure 5.7: Extending the Language

As shown in figure 5.7 above, users can simply type their definitions and these will be added to the language and stored in a map for future use. Adding the zero-coupon discount bond to the language, as shown above, again proves that complex contracts such as *zcb* can be defined by combining multiple simpler contracts, as we can now form the following contract: **zcb "24/12/2020 23:33:33" 200 andGive one**.

Chapter 6

Implementation

This chapter will discuss how the system was designed and what technical components were used for our implementation. Firstly, it will be discussed which technologies were utilised and why these are suitable options for this project. Following this, this chapter will go on to elaborate on the architectural system design and how different components interact with each other.

6.1 Technologies Used

6.1.1 WebAssembly

"WebAssembly (Wasm) is a binary instruction format for a stack-based virtual machine. Wasm is designed as a portable target for compilation of high-level languages like C, C++ or Rust, enabling deployment on the web for client and server applications" [54]. Since the launch of Wasm in 2017, developers are no longer required to write JavaScript code when creating web applications. Using Wasm as web development language enables developers to write code in their preferred language, but also opens access to a wider community of developers. Wasm is supported by all major browsers and easily extensible, meaning that developers can easily add new features, such as garbage collection, if needed. While using Wasm brings many advantages, its functions may also be imported and used within JavaScript, particularly to allow developers to take advantage of Wasm's performance enhancements.

Nowadays, there are several blockchain projects focusing on Wasm, including Ethereum, Parity and Dfinity. Using Wasm for this project implies that users are not required to write any Solidity smart contract code, since contracts can be pre-defined in a Rust script, which will prevent the security threats mentioned in section 2.4.3. Wasm also comes with its own virtual machine, further reducing smart contract vulnerabilities caused by the EVM. Security remains of top priority throughout the development of this project.

Furthermore, Wasm is known for its highly efficient performance thanks to its binary format, resulting in near-native speeds with execution times only 20% slower than na-

tive execution. In the blockchain world particularly, performance is of high priority. Wasm uses a sandboxed execution environment and enforces the same origin and permission security policies of the browser that it is executed in, while additionally making use of code obfuscation over the browser. This makes Wasm a highly efficient, secure and fully portable web environment suitable for blockchain development.

6.1.2 Rust

Using Wasm in a project allows the use of a wide range of programming languages that compile into Wasm. The list of languages directly compatible with Wasm (including Rust, Solidity, C, C++, Swift, PHP and many more) is exhaustive and ever-evolving. For this project Rust was used as the implementation language of a pre-defined smart contract. This contract comprises minimal functionality, focusing solely on storing the contract holder and counter party addresses, their corresponding balances, as well as on enabling them to transfer Ether between each other. This effectively reduces existing security threats by limiting our language and preventing human programming errors. Wasm is used to compile the pre-defined Rust contract into Wasm byte code, which is then further compiled into the final EVM code hash and JSON contract ABI. At this point, our contract code originally implemented in Rust has become deployable on the Ethereum blockchain.

Rust is a strongly-typed systems-level programming language similar to C, but it provides more compact data structures, making it particularly useful for the space-constrained blockchain world, as well as enhanced memory safety, stopping developers from introducing coding bugs and making it resistant to segmentation faults, while maintaining the performance levels of C. Rust is type-safe and has no run-time behaviour, such as garbage collection or reference counting, and it also has no undefined behaviour. Rust provides functional as well object-oriented programming features. Developers favour Rust because of its smart compiler, which outputs very useful compiler messages, thus simplifying the debugging process.

Since no Solidity code is required to be written by the end user in order for them to interact with Ethereum smart contracts, the security vulnerabilities present with Solidity development (see section 2.4.3) are eliminated.

6.1.3 Parity

Nowadays, there are multiple different Ethereum clients, or protocols, including Mist, Geth and Parity. Parity is an Ethereum startup company and said to be the world's fastest and lightest Ethereum client. Parity is written in Rust, providing enhanced

performance, security, reliability and code clarity [55], as mentioned in section 6.1.2. By using Wasm for their blockchain development, Parity is not to restricted to dealing with 256-bit integers, but this enables them to also work with 64-bit and 32-bit integers, providing Parity with the freedom to do more things and have lower level instructions. In addition to this, being based on Wasm will terminate the scalability problems of the EVM. Wasm optimises execution and load times and hence code can be processed more efficiently, resulting in faster transaction times. In Parity all implementations are fully deterministic and behave the same way.

6.1.4 eWasm & pWasm

Because of the EVM overhead mentioned in section 2.4.3, Ethereum are planning on replacing the standard EVM with eWasm: Ethereum-flavoured WebAssembly. This is one of the major objectives in the Ethereum 2.0 roadmap [56]. The main goals of eWasm are to enhance performance, hardware support and portability by replacing the EVM with a subset of Wasm instructions and adding relevant features and Wasm components, such as determinism.

The current EVM runs bytecode that does not map to any ordinary machine architecture because of its reliance on large 256-bit registers. This means that all data is handled with 256-bit precision even though it may not be needed. Therefore, even the simplest set of calculations needs to be transformed to be usable for the EVM, which turns out to be an extensive and costly process. Functions in the bytecode produced by Wasm, on the other hand, can be closely matched to hardware instructions, like Parity being able to deal with 64-bit and 32-bit integers, leading to significant performance enhancements. Generally speaking, eWasm is Wasm minus nondeterminism (such as floating points) plus metering and Ethereum environment interface methods.

Our system makes use of pWasm, which is very similar to eWasm and both of these projects are trying to achieve common goals. The only noticeable difference is that pWasm uses Parity as their Ethereum client (see section 6.1.3). To conclude, pWasm allows blockchain development to be more secure, faster, more portable and to have more tools available for development, while benefiting of the Wasm ecosystem.

6.2 Smart Contract Processing

Initally it was planned to implement a parser in Rust. Rust would have been an appropriate language because of the language's efficiency, strong typing and memory safety. These are prominent programming language features for producing fast and correct parsers. Additionally, as Peyton Jones et al. studied in their paper [1], a programming language with functional features is highly suitable for creating a domain-specific combinator library for financial smart contracts. However, the Rust programming language could not be used for the parsing of the user input in this project, because of certain restrictions introduced by pWasm.

Firstly, one cannot transfer strings between JavaScript and Rust code. Therefore, in order to process the user input from the website one would have to encode the string first, send the encoding to the Rust file, decode it within Rust, followed by calculating the results, encoding these and sending the encoding back to JavaScript for the JavaScript to decode and retrieve the results. This works using an encoding from a string into a vector of decimals, however, this is an arduous process and turns out to become noticeably inefficient when working with large input strings.

Furthermore, by relying on the pWasm libraries provided by the Rust language (*pwasm_std*, *pwasm_ethereum*, *pwasm_abi*, *pwasm_abi_derive*) inside the Rust smart contract code one restricts the contract file to being able to access pWasm libraries only. This means that if one wants to do any string manipulation, which is necessary for the parsing the user input, one is restricted to using the functions and types provided by the pWasm libraries. Parsing the user's input string requires the use of pattern matching and comparing dates to check for instance whether a contract has expired. Both of these require the use of external Rust libraries, which are not provided by pWasm itself. Rust parsers that can be found online, such as the arithmetic expressions parser written by Neumann [57], rely on some type of external library and therefore are not of use for this project.

Even if the code for the parsing is placed in another Rust file inside another folder the smart contract code will not compile successfully. This is because in order to use Rust to produce Wasm code, Rust relies on a build tool called Cargo. Cargo makes use of a Cargo.toml file that lists all dependencies required for the project. Therefore, there will be a dependency issue present whenever the project tries to use a pWasm library and the Rust standard *std* library for the same project.

Because of the reasons mentioned above, this project is processing the user input using a JavaScript parser and Rust is used purely for the smart contract implementation. The implemented parser analyses the user input and resolves it into its logical syntactic components. These components, or lower level smart contracts, are then evaluated and

executed.

6.2.1 Conditionals

Although Peyton Jones et al. touch on the topic of comparisons and mention their development of a notion of ordering between observables as well as contracts [1], they do not include conditionals in their language grammar or their Haskell implementation. The authors confirm that such comparisons can be used for transformation optimisations in the valuation process in order to allow contracts to be "valued more cheaply". In general, conditionals alter the control flow based on some condition and they allow us to compare two contracts in terms of their value, horizon or domination and give parties a choice regarding the output of such comparison. For instance, if contract c_1 is worth more than c_2, then the contract owner may want to execute some other contract c_3. Nexus implements such conditionals with if-statements that allow users to compare the horizon, the value (in Ether), or the domination of two contracts, further extending our context-free grammar from section 5.4. The basic syntax of a conditional statement in our language is the following:

$$
\begin{aligned}
&if \;\; \textit{boolean condition} \\
&\qquad \textit{consequent} \\
&\{\; else \\
&\qquad \textit{alternative} \;\}
\end{aligned}
$$

This is a common syntax for conditionals used in programming languages like C++, Java and JavaScript. Before a contract string is broken down into its subcontracts, our program parses through the contract string and evaluates every if clause, replacing it either by its consequent or its alternative (depending on the value of its condition). The conditional expression of the clause, *condition*, must be a boolean value that evaluates to either true or false. If this value evaluates to true, the *consequent* is executed, otherwise, the *alternative* is executed. While the *condition* and the *consequent* must be present, the *else* and *alternative* are both optional. If the *condition* evaluates to true, but there is no *else* clause or *alternative* present, the conditional statement is replaced by the empty string. Our language also allows us to combine more than one condition into a single condition using the logical operators || ("or") and && ("and"), as long as the compositional condition reduces to a single boolean value. By extending our language with conditionals that may contain *alternative* statements, our context-free grammar from section 5.4 becomes ambiguous. This means that the conditional state-

ment **if cond$_A$ then if cond$_B$ then cons$_C$ else alt$_D$** can be parsed as **if cond$_A$ then (if cond$_B$ then cons$_C$) else alt$_D$** as well as **if cond$_A$ then (if cond$_B$ then cons$_C$ else alt$_D$)**. Thus, there is more than one correct parse tree for the given contract string. This is known as the Dangling Else Ambiguity problem [58]. Our language resolves this problem by requiring users to place curly parenthesis around the consequent and the alternative of any conditional statement. Additionally, our language requires conditions to be parenthesised. This creates explicit blocks for any conditional statement, their consequent and alternative and effectively, these enforcements delimit every conditional block.

In order to distinguish between the horizon and value comparisons, Nexus uses the usual comparison operators ($==, >, \geq, <, \leq$) contained inside curly brackets for horizon comparisons and the same operators contained inside square brackets for value comparisons. In order to check whether one contract dominates another, the comparison operators are used without any padding. What is fundamentally important is the fact that contracts are compositional (see section 6.2.2). Therefore, the horizon and value of a contract depend on the horizon and value of its subcontracts. By extending our language with conditional statements, we can now construct a final representation of our context-free grammar:

$$
\begin{aligned}
SUPER\_CONT &\rightarrow SUPER\_CONT\ CONJ\ SUPER\_CONT \mid SUB\_CONT \\
CONJ &\rightarrow and \mid or \\
SUB\_CONT &\rightarrow if(BOOL)\ \{CONJ\_CONT\}\ else\ \{CONJ\_CONT\} \mid if(BOOL)\ \{CONJ\_CONT\}\ else\ \{\varepsilon\} \\
&\quad \mid if(BOOL)\ \{CONJ\_CONT\} \mid CONT \\
CONJ\_CONT &\rightarrow CONJ\_CONT\ CONJ\ CONJ\_CONT \mid CONT \\
BOOL &\rightarrow BOOL\ LOG\_OP\ BOOL \mid SUPER\_CONT\ COM\_OP\ SUPER\_CONT \\
LOG\_OP &\rightarrow \| \mid \&\& \\
COM\_OP &\rightarrow > \mid < \mid \geq \mid \leq \mid == \mid \{>\} \mid \{<\} \mid \{\geq\} \mid \{\leq\} \mid \{==\} \mid [>] \mid [<] \mid [\geq] \mid [\leq] \mid [==] \\
CONT &\rightarrow scaleK\ NUM(CONT) \mid get(CONT) \mid truncate\ DATE(CONT) \mid give(CONT) \mid TERM \\
TERM &\rightarrow one \mid zero \\
NUM &\rightarrow ([1-9]\ DIG*(\ .\ DIG*[1-9])?) \mid (0\ (\ .\ DIG*[1-9])?) \\
DIG &\rightarrow [0-9] \\
DATE &\rightarrow ((0?\ [1-9]) \mid ([12]\ DIG) \mid (3\ [01])) / ((0?\ [1-9]) \mid (1\ [0-2])) / (DIG\ DIG\ DIG\ DIG) \\
&\quad \backslash s\ ((0\ DIG) \mid (1\ DIG) \mid (2\ [0-3])) : ([0-5]\ DIG) : ([0-5]\ DIG)
\end{aligned}
$$

Figure 6.1: Final Representation of the Language Grammar

6.2.1.1 Horizon

The horizon H of a contract is defined as the latest possible time at which the contract can be acquired. After this time has passed, the contract will have expired. By default, the horizon of a contract is infinite, however, if a low-level contract includes the *truncate* combinator, the contract horizon changes. For a contract c_1 defined as **truncate t $\mathbf{c_2}$**, c_1's horizon becomes the smaller of t and c_2's original horizon. Naturally, the horizon of a conjunction or disjunction contract, such as **$\mathbf{c_1}$ and $\mathbf{c_2}$** and **$\mathbf{c_1}$ or $\mathbf{c_2}$**, is always the maximum horizon out of its subcontracts **$\mathbf{c_1}$** and **$\mathbf{c_2}$**. The remaining horizon definitions are given in the table below.

H(zero)	=	infinite
H(one)	=	infinite
H(give c)	=	H(c)
H(scaleK k c)	=	H(c)
H(truncate t c)	=	min(t, H(c))
H(get c)	=	H(c)
H(c_1 and c_2)	=	max(H(c_1), H(c_2))
H(c_1 or c_2)	=	max(H(c_1), H(c_2))

Figure 6.2: Horizon Definition [1]

Algorithm 1 computes the horizon for any given contract string S. It recursively calculates the horizon of every subcontract by finding its minimum horizon, while keeping track of the maximum horizon of all subcontracts. This maximum horizon will then be returned as the final horizon of the input contract. If the algorithm finishes iterating through a subcontract that does not contain the *truncate* combinator, implying that it possesses an infinite horizon, we can safely terminate the process and conclude that the input contract's horizon is infinite.

Algorithm 1 Getting the Horizon of a Contract

```
1: procedure GetHorizon(S)                          ▷ The horizon of contract string S
2:     maxHor ← ε
3:     comeAcrossTruncate ← false
4:     for every word W ∈ S do
5:         if W = truncate then
6:             currentHor ← next word
7:                 ▷ iterate until a balanced closing parenthesis or one/zero is found
8:             c ← obtainSubContract(S.slice(current index+2))
9:             prevHor ← GetHorizon(c)                ▷ obtain c's previous horizon
10:            if greaterDate(currentHor, prevHor) then
11:                currentHor ← prevHor
12:            end if
13:            comeAcrossTruncate ← true
14:            if maxHor = ε ∨ greaterDate(currentHor, maxHor) then
15:                maxHor ← currentHor
16:            end if
17:            i += c.length
18:        else if W = and ∨ W = or then              ▷ reached subcontract end
19:            if ¬comeAcrossTruncate then
20:                return "infinite"                  ▷ The contract horizon is infinite
21:            end if
22:            comeAcrossTruncate ← false
23:        else if index = S.length ∧ ¬comeAcrossTruncate then
24:            return "infinite"                      ▷ The contract horizon is infinite
25:        end if
26:    end for
27:    return maxHor
28: end procedure
```

6.2.1.2 Value

When reasoning about the value V of a conjunction or disjunction contract, things become a little more tricky. This is because the value of such a contract depends on the horizon of each subcontract. Take for example the contract $\mathbf{c = c_1}$ **and** $\mathbf{c_2}$. At first sight one would think that the value of such a compositional contract is the sum of both values $V(c_1)$ and $V(c_2)$. Indeed, this is the case if the current time is before the horizon of both c_1 and c_2. Therefore, the value of **one and scaleK 100 (give one)** is -99 Ether, as both subcontracts possess an infinite horizon. A negative contract value

implies that the contract owner is paying the specified amount to the counter-party. However, if for instance c_1 has expired, the value of c will solely be the value of c_2. Similarly, for $\mathbf{c = c_1 \text{ or } c_2}$, *V(c)* will be the maximum of $V(c_1)$ and $V(c_2)$. However, when c_1 has expired, *V(c)* will solely be $V(c_2)$. Therefore, we can say that the value of a contract is a function dependent on the time for which this value is requested. While Peyton Jones et al. [1] make the value of a *get* contract dependent on interest rates, we say that such a contract only possesses a value at the time (or day) of its horizon. This makes sense, because *get c* is acquired at its horizon and the contract holder does not have the choice to acquire it any time before then. Hence, c does not have any value to either party before or after *H(c)*. The remaining value definitions illustrated in the table given below are rather straightforward.

V(zero)	=	0	
V(one)	=	1	
V(give c)	=	-V(c)	
V(scaleK k c)	=	k * V(c)	
V(truncate t c)	=	V(c)	on $\{t \mid t \leq H(c)\}$
		0	on $\{t \mid t > H(c)\}$
V(get c)	=	V(c)	on $\{t \mid t = H(c)\}$
		0	on $\{t \mid t > H(c) \wedge t < H(c)\}$
$V(c_1 \text{ and } c_2)$	=	$V(c_1) + V(c_2)$	on $\{t \mid t \leq H(c_1) \wedge t \leq H(c_2)\}$
		$V(c_1)$	on $\{t \mid t \leq H(c_1) \wedge t > H(c_2)\}$
		$V(c_2)$	on $\{t \mid t > H(c_1) \wedge t \leq H(c_2)\}$
$V(c_1 \text{ or } c_2)$	=	$\max(V(c_1), V(c_2))$	on $\{t \mid t \leq H(c_1) \wedge t \leq H(c_2)\}$
		$V(c_1)$	on $\{t \mid t \leq H(c_1) \wedge t > H(c_2)\}$
		$V(c_2)$	on $\{t \mid t > H(c_1) \wedge t \leq H(c_2)\}$

Figure 6.3: Value Definition [1]

In order to compute the value V for any contract string S, algorithm 2 was implemented. This consists of two sub-algorithms: *GetValue* and *GetUnderlyingValue*, where the latter is a simple function called by *GetValue* on low level contracts containing a single connective only. For efficiency purposes, *GetValue* makes use of a stack, *combinatorStack*, that keeps track of combinators to be applied to lower-level contracts. Whenever the loop reads an opening parenthesis we want to push the combinators that are to be applied to what is following after the parenthesis on top of the stack. These combinators can then be popped off the stack and applied to the currently parsed string, *currentStr*, when the loop reads a closing parenthesis. In the special case when the contract string includes the *get c* combinator, the value of c needs to be computed and it is checked whether the day of the horizon of c corresponds to the day of the

current date. If so, the value of c is returned, else we return 0 as the value for the *get* c part of the contract.

Algorithm 2 Getting the Value of a Contract

```
1: procedure GetValue(S)                          ▷ The value of contract string S
2:     currentStr ← ε
3:     combinatorStack ← []
4:     currentVal
5:     for every word W ∈ S do
6:         if W = ( then
7:             if currenStr ≠ ε then
8:                 combinatorStack.push(currentStr)
9:                 currentStr ← ε
10:            end if
11:        else if W =) then          ▷ evaluate currentStr, pop off combinatorStack
12:            currentVal ← GetUnderlyingValue(currentStr)
13:            if combinatorStack.length > 0 then
14:                str ← combinatorStack.pop + currentVal
15:                currentVal ← GetUnderlyingValue(str)
16:            end if
17:            if currentVal ≠ undefined then
18:                if end of loop then
19:                    currentStr ← ε
20:                else
21:                    currentStr ← currentVal
22:                end if
23:            end if
24:        else if W = get then                                  ▷ special case
25:            c ← obtainSubContract(S.slice(current index+1))
26:            tempVal ← GetValue(c)
27:            if GetHorizon(c) is same day as current date then
28:                currentVal ← tempVal
29:            else
30:                currentVal ← 0
31:            end if
32:            if currentVal ≠ undefined then
33:                if end of loop then
34:                    currentStr ← ε
35:                else
36:                    currentStr ← currentVal
37:                end if
```

```
38:             end if
39:             i += c.length
40:         else
41:             currentStr += W
42:         end if
43:     end for
44:     if currentStr ≠ ε then
45:         currentVal ← getUnderlyingVal(currentStr)
46:     end if
47:     return currentVal
48: end procedure
```

The subalgorithm *GetUnderlyingValue* is called by algorithm 2 whenever we have reached a subcontract that contains a single connective only. This is a trivial procedure that firstly checks the type of the connective. Following this, *GetUnderlyingValue* is re-called on the two subcontracts separated by the connective and if the connective corresponds to *and*, we return the sum of the two results, otherwise the maximum value of the two results is returned, adhering to the definitions in figure 6.3. When *GetUnderlyingValue* is called on a lowest-level contract containing no connective, it simply iterates through the contract keeping track of the current value initialised to 1, which is multiplied when *scaleK* or an observable is read and negated when *give* is read. When reading a number, the number itself can be returned as the value. At the end of the loop the computed value is returned. The function returns 0 if the horizon of this contract has expired or if the contract string includes *zero*.

To clarify the behaviour of this algorithm we present a simulation on the contract **(one or (one and scaleK 10(one))) and truncate "24/12/2017 23:33:33"(one)** in figure 6.4. Note that in step 18 we calculate the value of **one**, which corresponds to **1**. Following this, since *combinatorStack* is non-empty, we pop its top element and request the value of **11 and truncate "24/12/2017 23:33:33" 1** and since the second subcontract has expired, *GetUnderlyingValue* will return **0** for its value and **11+0** equals **11**. If the horizon of the second subcontract was some date in the future, we would returns the sum of its value and **11**.

Step	currentStr	combinatorStack	currentVal	Unparsed
0	ε	[]		(one or (one and scaleK 10(one))) and truncate “24/12/2017 23:33:33”(one)
1	(	[]		one or (one and scaleK 10(one))) and truncate “24/12/2017 23:33:33”(one)
2	(one	[]		or (one and scaleK 10(one))) and truncate “24/12/2017 23:33:33”(one)
3	(one or	[]		(one and scaleK 10(one))) and truncate “24/12/2017 23:33:33”(one)
4	ε	[(one or]		one and scaleK 10(one))) and truncate “24/12/2017 23:33:33”(one)
5	one	[(one or]		and scaleK 10(one))) and truncate “24/12/2017 23:33:33”(one)
6	one and	[(one or]		scaleK 10(one))) and truncate “24/12/2017 23:33:33”(one)
7	one and scaleK	[(one or]		10(one))) and truncate “24/12/2017 23:33:33”(one)
8	one and scaleK 10	[(one or]		(one))) and truncate “24/12/2017 23:33:33”(one)
9	ε	[(one or, one and scaleK 10]		one))) and truncate “24/12/2017 23:33:33”(one)
10	one	[(one or, one and scaleK 10]		))) and truncate “24/12/2017 23:33:33”(one)
11	11	[(one or]	11	)) and truncate “24/12/2017 23:33:33”(one)
12	11	[]	11	) and truncate “24/12/2017 23:33:33”(one)
13	11	[]	11	and truncate “24/12/2017 23:33:33”(one)
14	11 and	[]	11	truncate “24/12/2017 23:33:33”(one)
15	11 and truncate	[]	11	“24/12/2017 23:33:33”(one)
16	11 and truncate “24/12/2017 23:33:33”	[]	11	(one)
17		[11 and truncate “24/12/2017 23:33:33”]	11	one)
18	one	[11 and truncate “24/12/2017 23:33:33”]	11	)
19		[]	11	

Figure 6.4: Simulation of the GetValue Algorithm

The procedure of getting the value of a contract is efficient in the input string S, as *GetValue* iterates through S once only. Even in the special case when the input contract includes the *get c* combinator, although the algorithm will call *GetValue* recursively in order to compute the underlying value, this is optimised by skipping the indices that are iterated over during this process, effectively looping over every index once only. However, *getUnderlyingValue* loops through its input string, which was initially popped off *combinatorStack* in *GetValue* and hence iterated over previously, a second time in order to split this string by its connective. This adds an overhead to the overall value computation, which could be mitigated by separating *currentStr* by its connective in *GetValue* on-line and making use of a *combinatorStack* that stores triplet arrays containing the first subcontract, the connective, and the second subcontract of the compositional contract. This improvement should be considered for future versions of Nexus.

6.2.1.3 Domination

Peyton Jones et al. [1] define a notion of ordering between observables and contracts referred to as dominance. While they apply the $\geq$ operator to two contracts to describe dominance, we expand on this by using their notion to define strict dominance using the $>$ operator, as well as equality between contracts specified by the $==$ operator. A contract c_1 is said to dominate (non-strictly) contract c_2, that is $c_1 \geq c_2$, if and only if the horizon of c_1 is greater than or equal to the horizon of c_2 and the value of c_1 is greater than or equal to the value of c_2 at all times before c_1's horizon.

$$c_1 \geq c_2 \iff H(c_1) \geq H(c_2) \wedge \forall t \leq H(c_2).V(c_1)(t) \geq V(c_2)(t) \tag{6.1}$$

We use the definition above to deduce the following additional operators.

$$c_1 > c_2 \iff H(c_1) > H(c_2) \wedge \forall t < H(c_2).V(c_1)(t) > V(c_2)(t) \tag{6.2}$$
$$c_1 \leq c_2 \iff H(c_1) \leq H(c_2) \wedge \forall t \leq H(c_1).V(c_1)(t) \leq V(c_2)(t) \tag{6.3}$$
$$c_1 < c_2 \iff H(c_1) < H(c_2) \wedge \forall t < H(c_1).V(c_1)(t) < V(c_2)(t) \tag{6.4}$$
$$c_1 == c_2 \iff H(c_1) == H(c_2) \wedge \forall t \leq H(c_2).V(c_1)(t) == V(c_2)(t) \tag{6.5}$$

This allows us to directly compare two contracts and introduce a formal ordering between them. Additionally, these identities can be applied in the evaluation process

(section 6.2.1.4), as well as to optimise transformations into an intermediate contract representation that will allow for faster contract processing (section 6.2.4).

The algorithm used for evaluating dominance between two contracts, c_1 and c_2, iterates through every occurring horizon time in both contracts and checks whether $V(c_1)$ is greater than or equal to $V(c_2)$ (for the $\geq$ operator). If at any of those times the requirement is not met, the algorithm returns false, formula 6.1 does not hold and c_1 does not dominate c_2. Otherwise, c_1 indeed dominates c_2. For instance, evaluating **(truncate "20/12/2017 23:33:33"(one) and truncate "24/12/2017 23:33:33"(zero)) == truncate "24/12/2017 23:33:33"(one)** will firstly check whether the horizon of the left-hand side contract equals the horizon of the right-hand side contract. Because this is indeed the case, we check at 20/12/2017 23:33:33 whether the value of the first contract corresponds to the value of the second contract. Again, the condition is met since both values are equal to 1. We proceed to the next occurring horizon time: 24/12/2017 23:33:33. At this time, however, the value of the left-hand side contract equals 0, whereas the value of the right-hand side contract is still 1, so our algorithm returns false.

6.2.1.4 Conditional Evaluation

When parsing the contract input string provided by the user, first of all, if-statements need to be detected, evaluated and replaced by their respective outcome. In order to evaluate conditionals we present the *EvaluateConditionals* parsing algorithm that was implemented alongside the functions mentioned in previous sections. Algorithm 3 presents the pseudocode of this process. Fundamentally, the program iterates through the contract string word by word to detect each if-condition, their consequence, their alternative (if present), but also keeps track of the combinators that are to be applied to the result. This makes this process complex, yet Algorithm 3 efficiently solves this problem by looping through the contract string once only, thus resulting in a linear runtime. This complexity is achievable by making use of three different stacks. Firstly, *noOfOpeningParensStack* is used to keep track of the difference between the number of opening parenthesis seen, *openingParens*, and the number of closing parenthesis seen, *closingParens*, and adds this difference to the stack every time the algorithm reads "if", while incrementing the *ifsToBeMatched* counter. When reading a closing parenthesis in line 20 of the algorithm, this stack is then used to check whether we have reached the end of an if-condition and if so, the top element of the stack can be popped off at the end of the iteration.

The second stack that we use is called *firstPartStack*. When a comparison operator is captured, our algorithm pushes the current value of *ifCondition*, a string variable used to keep track of the currently parsed if-condition of the statement, to the top of this stack, and when reaching the end of an if-condition we pop the top element off the

stack to evaluate the boolean value of the condition together with the corresponding comparison operator and second part of the if-condition. The top of this stack is also popped when iterating over a boolean operator, such as && or ||, as seen in lines 85-87 of the pseudocode. This is to allow for efficient evaluation of the first input into the boolean operator on the go by applying the following elementary logic rules:

$$true \wedge B \equiv B \tag{6.6}$$
$$false \vee B \equiv B \tag{6.7}$$

Making use of these identities further optimises the performance of the algorithm. The third and final stack that is used to efficiently evaluate conditionals in our algorithm is *compOpStack*. This simply accumulates comparison operators throughout the contract string and pops these when calling the *Evaluate* function to retrieve the boolean value of the current if-condition. This complex, yet efficient implementation also makes use of syntax checking to notify users when composing contracts that are syntactically invalid. This has been left out in the pseudocode represented below.

Algorithm 3 Evaluating Conditionals

```
1: procedure EvaluateConditionals(S)   ▷ Evaluation of contract conditionals
2:     openingParens ← 0
3:     closingParens ← 0
4:     ifCondition ← ε
5:     contractString ← ε
6:     noOfOpeningParensStack ← []
7:     firstPartStack ← []
8:     ifsToBeMatched ← 0
9:     insideCondition ← false
10:    compOpStack ← []
11:    for every word W ∈ S do
12:        if W = if then
13:            noOfOpeningParensStack.push(openingParens − closingParens)
14:            ++ifsToBeMatched
15:            insideCondition ← true
16:        else if W = ) then
17:            ++closingParens
18:            bool1 ← noOfOpeningParensStack.length = 0 ∧ openingParens =
    closingParens ∧ ifsToBeMatched ≠ 0
19:            bool2 ← openingParens−noOfOpeningParensStack.top = closingParens
```

```
20:         if bool1 ∨ bool2 then
21:             insideCondition ← false
22:             firstPart ← firstPartStack.pop
23:             compOp ← compOpStack.pop
24:             ifCondition ← ε
25:             bool ← evaluate(firstPart, compOp, ifCondition)
26:             cons ← findConsequent(S, index+2)                ▷ parses up to }
27:             if bool then
28:                 if ifsToBeMatched > 1 then
29:                     ifCondition ← cons
30:                 else
31:                     contractString += cons
32:                 end if                          ▷ can now skip indices to fast forward
33:                 if word after cons.length + 2 ≠ else then
34:                     index += cons.length + 1
35:                 else
36:                     alt ← findConsequent(S, index+cons.length + 4)
37:                     index += cons.length + alt.length + 3
38:                 end if
39:             else                                                  ▷ bool is false
40:                 if word after cons.length + 2 ≠ else then
41:                     index += cons.length + 1
42:                     if ifsToBeMatched > 1 then
43:                         ifCondition ← ε
44:                     else                ▷ append result (nothing) to contractString
45:                         if last word in contractString is connective then
46:                             contractString ← contractString.slice(0, index of connective)
47:                         end if
48:                     end if
49:                     if next word is connective then
50:                         ++index
51:                     end if
52:                 else                                    ▷ there is an alternative
53:                     alt ← findConsequent(S, index+cons.length + 4)
54:                     index += cons.length + alt.length + 3
55:                     if ifsToBeMatched > 1 then
56:                         ifCondition ← alt
57:                     else contractString += alt
58:                     end if
59:                 end if
60:             end if
61:             − − ifsToBeMatched
```

```
62:                 noOfOpeningParensStack.pop
63:             else if ifsToBeMatched ≤ 0 ∧ ¬insideCondition then
64:                 contractString += W
65:             end if
66:         else if W = ( then
67:             ++ openingParens
68:             if prev word ≠ if then
69:                 if ifsToBeMatched ≤ 0 ∧ ¬insideCondition then
70:                     contractString += W
71:                 end if
72:             end if
73:         else if W is comp op then
74:             if firstPartStack.length = 0 ∨ firstPartStack.length ≠ ifsToBeMatched
    then
75:                 firstPartStack.push(ifCondition)
76:             end if
77:             compOpStack.push(W)
78:             ifCondition ← ε
79:         else if W is bool op then
80:             firstPart ← firstPartStack.pop
81:             compOp ← compOpStack.pop
82:             secondPart ← ifCondition
83:             ifConditionalVal ← evaluate(firstPart, compOp, secondPart)
84:             ifCondition ← ifConditionalVal + W
85:             if (ifConditionalVal ∧ W = &&) ∨ (¬ifConditionalVal ∧ W = ||) then
    ▷ they cancel each other out
86:                 ifCondition ← ε
87:             end if
88:         else
89:             if ifsToBeMatched > 0 then
90:                 ifCondition += W
91:             else
92:                 contractString += W
93:             end if
94:         end if
95:     end for
96:     return contractString          ▷ The contract string with no more conditionals
97: end procedure
```

Once the whole if-condition, including its first part, its boolean operator and its second part, has been captured, this is used as input into the *Evaluate* function. This function simply makes use of the *GetHorizon* and *GetValue* functions defined in earlier sec-

tions to produce a boolean output based on the given comparison operator. The value of this boolean determines whether to replace the whole conditional statement by its consequent or alternative (if present) in the *contractString* value inside the *EvaluateConditionals* function. If the value returned by *Evaluate* is true, *EvaluateConditionals* will append the if-statement's consequent to the value of *contractString*, otherwise it will use its alternative (if present). If no alternative is given, but the boolean of the condition evaluates to false, the whole conditional clause is removed from the contract string and *EvaluateConditionals* continues iterating through the remainder of the contract. This makes sense, because $\varepsilon \wedge B = B$ and $\varepsilon \vee B = B$ can be assumed to be true.

In order to demonstrate and visualise the behaviour of the rather complex *EvaluateConditionals* algorithm with an example, we present a simulation of the process on the contract string **one and if(((if(zero**[>]**one) {zero} else {one})** [<] **truncate "24/03/2019-23:33:33"(one)) ‖ (zero**[<=]**one)) {zero} else {give(one)}** below.

Step	ifCond	conStr	NOPS	FPS	COPS	Unparsed
0	ε	ε	[]	[]	[]	one and if(((if(zero[>]one) {zero} else {one}) [<] truncate "24/03/2019-23:33:33"(one)) ‖ (zero[<=]one)) {zero} else {give(one)}
1	ε	one	[]	[]	[]	and if(((if(zero[>]one) {zero} else {one}) [<] truncate "24/03/2019-23:33:33"(one)) ‖ (zero[<=]one)) {zero} else {give(one)}
2	ε	one and	[]	[]	[]	if(((if(zero[>]one) {zero} else {one}) [<] truncate "24/03/2019-23:33:33"(one)) ‖ (zero[<=]one)) {zero} else {give(one)}
3	ε	one and	[0]	[]	[]	(((if(zero[>]one) {zero} else {one}) [<] truncate "24/03/2019-23:33:33"(one)) ‖ (zero[<=]one)) {zero} else {give(one)}
4	ε	one and	[0]	[]	[]	((if(zero[>]one) {zero} else {one}) [<] truncate "24/03/2019-23:33:33"(one)) ‖ (zero[<=]one)) {zero} else {give(one)}
5	ε	one and	[0]	[]	[]	(if(zero[>]one) {zero} else {one}) [<] truncate "24/03/2019-23:33:33"(one)) ‖ (zero[<=]one)) {zero} else {give(one)}
6	ε	one and	[0]	[]	[]	if(zero[>]one) {zero} else {one}) [<] truncate "24/03/2019-23:33:33"(one)) ‖ (zero[<=]one)) {zero} else {give(one)}
7	ε	one and	[0,3]	[]	[]	(zero[>]one) {zero} else {one}) [<] truncate "24/03/2019-23:33:33"(one)) ‖ (zero[<=]one)) {zero} else {give(one)}
8	ε	one and	[0,3]	[]	[]	zero[>]one) {zero} else {one}) [<] truncate "24/03/2019-23:33:33"(one)) ‖ (zero[<=]one)) {zero} else {give(one)}
9	zero	one and	[0,3]	[]	[]	[>]one) {zero} else {one}) [<] truncate "24/03/2019-23:33:33"(one)) ‖ (zero[<=]one)) {zero} else {give(one)}
10	ε	one and	[0,3]	[zero]	[[>]]	one) {zero} else {one}) [<] truncate "24/03/2019-23:33:33"(one)) ‖ (zero[<=]one)) {zero} else {give(one)}
11	one	one and	[0,3]	[zero]	[[>]]	) {zero} else {one}) [<] truncate "24/03/2019-23:33:33"(one)) ‖ (zero[<=]one)) {zero} else {give(one)}
19	one	one and	[0]	[]	[]	) [<] truncate "24/03/2019-23:33:33"(one)) ‖ (zero[<=]one)) {zero} else {give(one)}
20	one	one and	[0]	[]	[]	[<] truncate "24/03/2019-23:33:33"(one)) ‖ (zero[<=]one)) {zero} else {give(one)}
21	ε	one and	[0]	[one]	[[<]]	truncate "24/03/2019-23:33:33"(one)) ‖ (zero[<=]one)) {zero} else {give(one)}
22	truncate	one and	[0]	[one]	[[<]]	"24/03/2019-23:33:33"(one)) ‖ (zero[<=]one)) {zero} else {give(one)}
23	truncate "24/03/2019-23:33:33"	one and	[0]	[one]	[[<]]	(one)) ‖ (zero[<=]one)) {zero} else {give(one)}
24	truncate "24/03/2019-23:33:33"	one and	[0]	[one]	[[<]]	one)) ‖ (zero[<=]one)) {zero} else {give(one)}
25	truncate "24/03/2019-23:33:33" one	one and	[0]	[one]	[[<]]	)) ‖ (zero[<=]one)) {zero} else {give(one)}
26	truncate "24/03/2019-23:33:33" one	one and	[0]	[one]	[[<]]	) ‖ (zero[<=]one)) {zero} else {give(one)}
27	truncate "24/03/2019-23:33:33" one	one and	[0]	[one]	[[<]]	‖ (zero[<=]one)) {zero} else {give(one)}
28	ε	one and	[0]	[]	[]	(zero[<=]one)) {zero} else {give(one)}
29	ε	one and	[0]	[]	[]	zero[<=]one)) {zero} else {give(one)}
30	zero	one and	[0]	[]	[]	[<=]one)) {zero} else {give(one)}
31	εn	one and	[0]	[zero]	[[<=]]	one)) {zero} else {give(one)}
32	one	one and	[0]	[zero]	[[<=]]	)) {zero} else {give(one)}
33	one	one and	[0]	[zero]	[[<=]]	) {zero} else {give(one)}
34	ε	one and zero	[]	[]	[]	ε

Figure 6.5: Simulation of the EvaluateConditionals Algorithm

6.2.2 Contract Decomposition

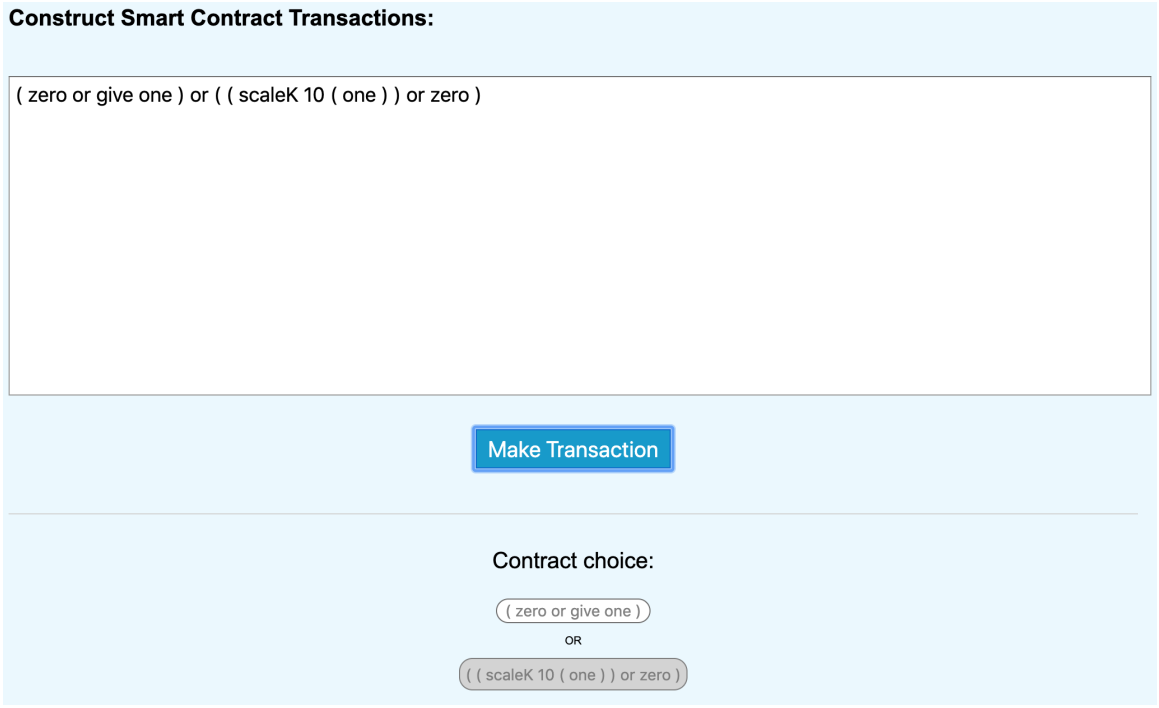


Figure 6.6: Contract Decomposition by Disjunction

After having evaluated and replaced conditional statements in the contract string, our program needs to decompose the contract into its lowest-level subcontracts. In addition to the combinators defined in section 5.4, there are two connectives that can be used to combine two subcontracts c_1, c_2: *and* and *or*. Using $\mathbf{c_1}$ **or** $\mathbf{c_2}$ in a contract gives the holder of the contract the choice to acquire either c_1 or c_2. The contract holder must make this choice before the chosen subcontract is made visible in the web application and added to the list of currently pending contracts. This software design choice was made in order to allow the web application to break down compositional contracts and display every lowest-level subcontract, including their contract string, their meaning translated into English, their horizon as well as their state (pending, successful, expired, insufficient funds), executable at their specific time of expiry. This ensures that usability remains of top priority throughout the project implementation. Otherwise, the system could only display the compositional contract and present the contract choice at the time of acquirement, reducing usablity of the system and the transparency of transactions. Secondly, the *and* connective can be applied to two contracts, c_1 and c_2, and will execute both underlying contracts upon acquirement.

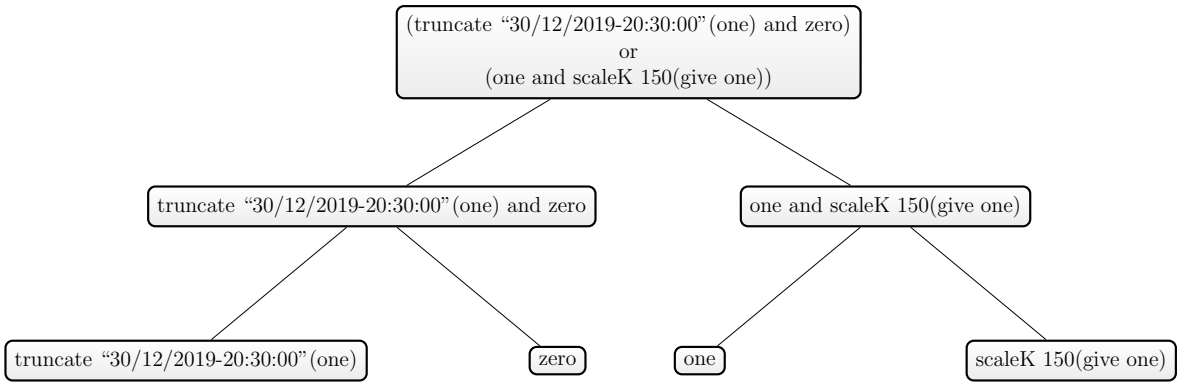


Figure 6.7: Subcontract Tree

The tree presented in the figure above shows how a compositional smart contract can be nested into multiple lowest-level subcontracts. At each parent node the tree splits the current contract string by its most balanced connective. If this connective corresponds to *or*, the user will be presented with a choice to pick one of the underlying subcontracts, as shown in figure 6.6. Otherwise, the algorithm continues splitting the two underlying subcontracts until reaching a leaf node. Once a node's contract string can no longer be split, that is it contains no more connectives, we have reached a leaf node and thus a lowest-level contract in our initial compositional contract representation.

For the decomposition of the input string by connectives, our program will need to find the 'outermost' occurrence of such a connective. By outermost, or highest-level, we refer to the connective that splits the current node in the tree into its two children nodes, or subcontracts. In order to effectively decompose a contract into its subcontracts, we make use of two algorithms: *Decompose* and *DecomposeByAnds*. The former is called when the contract string includes at least one occurrence of the *or* connective and effectively iterates over the whole contract string in order to split it into its two highest level subcontracts (conjuncts or disjuncts). This algorithm cannot simply split the contract string by every connective occurrence, because this will mess with the combinators surrounding a subcontract. For instance, let us look at the contract string **scaleK 10 (one or give one)**, where **scaleK 10** is applied to both **one** and **zero**. The naive algorithm implementation produces two subcontracts, namely **scaleK 10 one** and **give one**, rather than producing the expected result: **scaleK 10 one** and **scaleK 10 give one**. Therefore, we need to rely on a parsing algorithm that iterates through the contract string, while keeping track of all combinators that have been read and not yet applied, as well as the number of closing parentheses that will need to be added to each lowest-level subcontract. Our algorithm keeps track of the number of opening parenthesis read, *openingParens*, and the number of closing parenthesis read, named *closingParens*. When a connective is read it is checked whether *openingParens* equals *closingParens*. If so, we can be sure that we have found the

outermost connective and the two substrings to the left and right of the connective are the two highest-level subcontracts of the contract that is currently being decomposed. Furthermore, two additional variables are kept track of: *mostBalancedCon*, stating the type of the connective, and the minimal difference between *openingParens* and *closingParens*, namely *mostBalancedConBalance*. Whenever a connective is read and *openingParens* - *closingParens* is calculated to be smaller than the current value of *mostBalancedConBalance*, *mostBalancedConBalance* and *mostBalancedCon* are updated accordingly. If the outermost connective is *and* after having iterated through the entire contract string, the algorithm recurses and is automatically recalled on the two conjunct subcontracts. Otherwise, the user is presented with the contract choice, and after one of the options has been picked, the algorithm is recalled on the contract string of the picked choice. This process is repeated until no more connectives are present and the compositional contract can be displayed together with its subcontract components in the user interface of the web application.

The main components of this algorithm are three stacks, *parseStack*, *contractsStack* and *closingParensStack*, and *contractStr*, which keeps track of strings between two opening parentheses. The *parseStack* variable keeps track of all combinators that have been read but not yet matched to a lowest-level contract. Whenever an opening parenthesis is read we want to make a copy of the top element of that stack (assuming it is non-empty), append an opening parenthesis as well as *contractString* to it, and finally push it onto the stack. When a closing parenthesis or *and* is read, we can be certain that we have reached the end of a subcontract and can pop off the *parseStack* and the *closingParensStack* to construct the entire subcontract string. The *contractsStack* array is used to store and return the two highest-level subcontracts. Using the *Decompose* function described above in order to decompose a contract into its two highest-level subcontracts takes O(2n) time to execute in the worst-case, where n represents the number of words in the input string. This is because the function initially iterates through the string to detect whether it contains an *or* occurrence (and call *DecomposeByAnds* in this case) and takes an additional n iterations to complete.

Algorithm 4 presents the pseudocode of this decomposition process. It should be noted that for the sake of simplicity the handling of parenthesis is left out in the pseudocode algorithms presented in this section.

Algorithm 4 Input Decomposition

```
1: procedure DECOMPOSE(S)                          ▷ The decomposition of input string S
2:     strToAddToBeginning ← ε
3:     strToAddToEnd ← ε
4:     contractStr ← ε
5:     contractParsed ← ε
6:     openingParens ← 0
7:     closingParens ← 0
8:     mostBalancedCon ← ε
9:     mostBalancedConBalance ← length(S)
10:    parseStack ← []
11:    contractsStack ← []
12:    closingParensStack ← []
13:    conWaitingToBeMatched ← false
14:    secondPartStr ← ε
15:    for every word W ∈ S do
16:        if W = or ∨ W = and then
17:            if openingParens = closingParens then
18:                mostBalancedCon ← W
19:                contractsStack[0] ← contractParsed
20:                contractsStack[1] ← rest of S
21:                return [contractsStack[0], contractsStack[1], mostBalancedCon, ε, ε]
22:            else if openingParens − closingParens < mostBalancedConBalance
   then                                    ▷ found a new most balanced connective
23:                mostBalancedConBalance ← openingParens − closingParens
24:                mostBalancedCon ← W
25:                combStr ← parseStack.top
26:                closingParensStr ← closingParensStack.top
27:                if combStr ≠ undefined then
28:                    if mostBalancedCon = or then
29:                        contractsStack[0] ← contractStr
30:                        strToAddToBeginning ← combinatorStr + (
31:                        strToAddToEnd ← closingParensStr
32:                    else contractsStack[0] ← contractParsed + closingParensStr
33:                    end if
34:                else contractsStack[0] ← contractParsed
35:                end if
36:                contractStr ← ε
37:                conWaitingToBeMatched ← true
38:            else
39:                secondPartStr += W
```

```
40:                 contractStr += W
41:             end if
42:         else if W = zero ∨ W = one then
43:             combStr ← parseStack.top
44:             closingParensStr ← closingParensStack.top
45:             if conWaitingToBeMatched then secondPartStr += W
46:             end if
47:             contractStr += W
48:         else if W = ) then
49:             if ¬conWaitingToBeMatched then
50:                 combStr ← parseStack.pop
51:                 closingParensStr ← closingParensStack.pop
52:             end if
53:             bool1 ← openingParens−closingParens = mostBalancedConBalance
54:             bool2 ← openingParens−closingParens = mostBalancedConBalance
    + 1 ∧ first W = (
55:             bool ← bool1 ∨ bool2
56:             if secondPartStr ≠ ε ∧ conWaitingToBeMatched ∧ bool then
57:                 if closingParensStr = undefined then secondPartStr += W
58:                 else secondPartStr += closingParensStr
59:                 end if
60:                 if mostBalancedCon = or then
61:                     contractsStack[1] ← secondPartStr
62:                 else
63:                     contractsStack[1] ← strToAddToBeginning + combinatorStr +
    secondPartStr
64:                 end if
65:                 secondPartStr ← ε
66:             end if
67:             contractStr ← ε
68:             + + closingParens
69:         else if W = ( then
70:             + + openingParens
71:             if contractStr ≠ ε ∧ openingParens−closingParens < mostBalancedConBalance
    then
72:                 parseStack.push(parseStack.top + contractStr)
73:             end if
74:             if previous W ≠ and ∧  index ≠ 0 then
75:                 closingParensStack.push( closingParensStack.top+) )
76:             end if
77:             if conWaitingToBeMatched then
78:                 secondPartStr += W
79:             end if
```

```
80:             contractStr ← ε
81:         else
82:             if conWaitingToBeMatched then
83:                 secondPartString += W
84:             end if
85:             contractStr += W
86:         end if
87:         contractParsed += W
88:     end for
89:     return [contractsStack[0], contractsStack[1], mostBalancedCon,
90:                     strToAddToBeginning, strToAddToEnd]
91: end procedure
```

DecomposeByAnds is called when the contract string contains no more *or* connectives, and the string can now simply be split by every occurrence of and, appending their preceeding combinators. This algorithm works similarly to the *Decompose* function, using *parseStack* to keep track of the combinators to be appended to contracts while iterating through the string. Additionally, *DecomposeByAnds* makes use of an array named *finalContractsArr*, which will return all subcontracts contained in the input string. Algorithm 5 represents a formal definition of this deterministic LR (Left-to-right, Rightmost derivation in reverse) parsing algorithm [59] that produces a single correct parse tree. It successfully decomposes a contract string by its *and* conjuncts into its lowest-level subcontracts in linear runtime and eventually returns these in an array, ready to be added to the list of pending contracts. This algorithm takes O(n) time to execute in the worst case, because we are iterating through the input string once only. LR parsing algorithms typically make use of a lookahead of k symbols, where k is usually 1, in order to prevent backtracking and guessing. While our algorithm does not require such a lookahead, we access the previous term in the string in line 41 when we read an opening parenthesis.

Algorithm 5 Input Decomposition by 'and' Conjunctions

```
1: procedure DecomposeByAnds(S)        ▷ The decomposition of input string S
2:     openingParens ← 0
3:     contractStr ← ε
4:     parseStack ← []
5:     closingParensStack ← []
6:     resultArr ← []
7:     for every word W ∈ S do
8:         if W = and then
9:             if contractStr ≠ ε then
10:                if openingParens = 0 then
11:                    resultArr.push(contractStr)
12:                else if parseStack.length > 0 then
13:                    resultArr.push(parseStack.top + (+contractStr +
14: closingParensStack.top)
15:                else if closingParensStack.length > 0 then
16:                    resultArr.push(contractStr + closingParensStack.top)
17:                else
18:                    resultArr.push(contractStr)
19:                end if
20:                contractStr ← ε
21:            end if
22:        else if W =) then
23:            − − openingParens
24:            combinatorStr ← parseStack.pop()
25:            closingParensStr ← closingParensStack.pop()
26:            if contractStr ≠ ε then
27:                resultArr.push(combinatorStr + (+contractStr +
28: closingParensStr)
29:                contractStr ← ε
30:            end if
31:        else if W = ( then
32:            + + openingParens
33:            if contractStr ≠ ε then
34:                if parseStack.length > 0 then
35:                    parseStack.push(parseStack.top + (+contractStr)
36:                else
37:                    parseStack.push(contractStr)
38:                end if
39:                contractStr ← ε
40:            end if
```

```
41:             if previous word ≠ and ∧ index ≠ 0 then
42:                 closingParensStack.push(closingParensStack.top+))
43:             end if
44:         else
45:             contractStr += W
46:         end if
47:     end for
48:     if contractStr ≠ ε then
49:         resultArr.push(contractStr)
50:     end if
51:     return resultArr                ▷ The array containing all conjunction subcontracts
52: end procedure
```

The table in figure 6.8 shows the simulation of the *DecomposeByAnds* algorithm executing on the contract string **truncate "24/12/2019 23:33:33"(scaleK 10(one) and (scaleK 7(one and zero)))**, where columns CS and CPS represent *contractStr* and *closingParenthesisStack*, respectively. It should be noted that the algorithm adds a space before and after every parenthesis, so that when iterating through the contract string every parenthesis counts as a word.

Step	CS	parseStack	CPS	resultArr	Unparsed
0	ε	[]	[]	[]	truncate “24/12/2019 23:33:33”(scaleK 10(one) and (scaleK 7(one and zero)))
1	truncate	[]	[]	[]	“24/12/2019 23:33:33”(scaleK 10 (one) and (scaleK 7(one and zero)))
2	truncate “24/12/2019 23:33:33”	[]	[]	[]	(scaleK 10 (one) and (scaleK 7 (one and zero)))
3	ε	[truncate “24/12/2019 23:33:33”]	[)]	[]	scaleK 10 (one) and (scaleK 7 (one and zero)))
4	scaleK	[truncate “24/12/2019 23:33:33”]	[)]	[]	10 (one) and (scaleK 7(one and zero)))
5	scaleK 10	[truncate “24/12/2019 23:33:33”]	[)]	[]	(one) and (scaleK 7 (one and zero)))
6	ε	[truncate “24/12/2019 23:33:33”, truncate “24/12/2019 23:33:33”(scaleK 10]	[),))]	[]	one) and (scaleK 7 (one and zero)))
7	one	[truncate “24/12/2019 23:33:33”, truncate “24/12/2019 23:33:33”(scaleK 10]	[),))]	[]	) and (scaleK 7 (one and zero)))
8	ε	[truncate “24/12/2019 23:33:33”]	[)]	[truncate “24/12/2019 23:33:33”(scaleK 10(one))]	and (scaleK 7 (one and zero)))
9	ε	[truncate “24/12/2019 23:33:33”]	[)]	[truncate “24/12/2019 23:33:33”(scaleK 10(one))]	(scaleK 7 (one and zero)))
10	ε	[truncate “24/12/2019 23:33:33”]	[)]	[truncate “24/12/2019 23:33:33”(scaleK 10(one))]	scaleK 7 (one and zero)))
11	scaleK	[truncate “24/12/2019 23:33:33”]	[)]	[truncate “24/12/2019 23:33:33”(scaleK 10(one))]	7 (one and zero)))
12	scaleK 7	[truncate “24/12/2019 23:33:33”]	[)]	[truncate “24/12/2019 23:33:33”(scaleK 10(one))]	(one and zero)))
13	ε	[truncate “24/12/2019 23:33:33”, truncate “24/12/2019 23:33:33”(scaleK 7]	[),))]	[truncate “24/12/2019 23:33:33”(scaleK 10(one))]	one and zero)))
14	one	[truncate “24/12/2019 23:33:33”, truncate “24/12/2019 23:33:33”(scaleK 7]	[),))]	[truncate “24/12/2019 23:33:33”(scaleK 10(one))]	and zero)))
15	ε	[truncate “24/12/2019 23:33:33”, truncate “24/12/2019 23:33:33”(scaleK 7]	[),))]	[truncate “24/12/2019 23:33:33”(scaleK 10(one)), truncate “24/12/2019 23:33:33”(scaleK 7(one))]	zero)))
16	zero	[truncate “24/12/2019 23:33:33”, truncate “24/12/2019 23:33:33”(scaleK 7]	[),))]	[truncate “24/12/2019 23:33:33”(scaleK 10(one)), truncate “24/12/2019 23:33:33”(scaleK 7(one))]	)))
17	ε	[truncate “24/12/2019 23:33:33”]	[)]	[truncate “24/12/2019 23:33:33”(scaleK 10(one)), truncate “24/12/2019 23:33:33” (scaleK 7(one)), truncate “24/12/2019 23:33:33”(scaleK 7(zero))]	))
18	ε	[]	[]	[truncate “24/12/2019 23:33:33”(scaleK 10(one)), truncate “24/12/2019 23:33:33”(scaleK 7(one)), truncate “24/12/2019 23:33:33”(scaleK 7(zero))]	)
19	ε	[]	[]	[truncate “24/12/2019 23:33:33”(scaleK 10(one)), truncate “24/12/2019 23:33:33”(scaleK 7(one)), truncate “24/12/2019 23:33:33”(scaleK 7(zero))]	

Figure 6.8: Simulation of Contract Decomposition by ’and’ Conjunctions

The use of *DecomposeByAnds* allows us not to have to iterate through the remaining

contract string more than once in order to extract all lowest-level subcontracts. For instance, if we were to call *Decompose* on $\mathbf{c_1}$ **and** ($\mathbf{c_2}$ **and** ($\mathbf{c_3}$ **and** $\mathbf{c_4}$)) our program would iterate through the contract string three times, whereas when using *Decompose-ByAnds* instead, it only takes a single loop iteration, limiting the efficiency overhead introduced by the *Decompose* algorithm.

To conclude, the time complexity of using *Decompose* to break down a contract string into its lowest-level subcontracts depends on the number of *and* connectives the input string contains. This is because *Decompose* recurses as long as the contract string contains such a conjunction in order to allow for the presentation of the contract choice in the web application. Therefore, it can be stated that there exists a trade-off between performance and usability and we have sacrificed some of the contract processing performance in order to allow for the illustrative representation of complex compositional contracts to the end-user, enhancing transaction transparency and allowing non-technical end users to understand more about contracts and the evolution of their transactions.

Having defined notions of ordering between contracts and all algorithms necessary for the parsing of arbitrary Nexus contracts, in the following section we will deduce several properties of our language that can be made use of for the final step of our contract processing procedure.

6.2.3 Language Identities

Designing Nexus from a set of accurately defined combinators and the introduction of previously described notions of ordering between contracts (section 6.2.1) creates a semantics for our language, allowing us to prove language properties, such as what it means for two contracts to be the same. The following set of language identities can be applied to Nexus contracts to reduce compositional contracts into equivalent simpler forms.

$$get(get(c)) \equiv get(c) \tag{6.8}$$
$$give(give(c)) \equiv c \tag{6.9}$$
$$give(c_1 \text{ } or \text{ } c_2) \not\equiv give(c_1) \text{ } or \text{ } give(c_2) \tag{6.10}$$
$$scaleK \text{ } K(c_1 \text{ } or \text{ } c_2) \not\equiv scaleK \text{ } K(c_1) \text{ } or \text{ } scaleK \text{ } K(c_2) \tag{6.11}$$

The proof for identity 6.8 is given in 6.12, where the time of acquirement t_i equals the horizon of the underlying contract *H(c)*, and 6.13, where t_i is different from *H(c)*.

$$
\begin{aligned}
&V(get(get(c))),\ where\ t_1 = t_2 = H(c)\\
&= V(get(get(c)))(t_1)\\
&= V(get(V(get(c))(t_2)))(t_1)\\
&= V(get(V(get(c))(t_2)))(t_2)\\
&(underlying\ contract\ is\ acquired\ at\ singe\ time\ instance)\\
&= V(get(V(c)))(t_2)\\
&= V(c)\\
&= V(get(c))(t_2)\\
&= V(get(c))
\end{aligned}
\tag{6.12}
$$

$$
\begin{aligned}
&V(get(get(c))),\ where\ t_1 = t_2\ \wedge\ t_{\mathrm{i}} \neq H(c)\\
&= V(get(get(c)))(t_1)\\
&= V(get(V(get(c))(t_2)))(t_1)\\
&= V(get(V(get(c))(t_2)))(t_2)\\
&(underlying\ contract\ is\ acquired\ at\ singe\ time\ instance)\\
&= V(get(0))(t_2)\\
&= 0\\
&= V(get(c))(t_2)\\
&= V(get(c))
\end{aligned}
\tag{6.13}
$$

The proof for identity 6.9 is trivial, because reversing the rights and obligations of a contract **give c** will obtain contract **c** in its original settings. Property 6.10 follows from our value definitions given in figure 6.3 and the fact that if *V(c₁)* and *V(c₂)* are different, *-max(V(c₁), V(c₂))* and *max(-V(c₁), -V(c₂))* will also be distinct. The same logic applies for the proof of language identity 6.11, where *scaleK K (max(V(c₁), V(c₂)))* will be different from *max(scaleK K (V(c₁)), scaleK K (V(c₂)))* for two distinct contract values *V(c₁)*, *V(c₂)*.

6.2.4 Compilation Into IR

The final part of the contract parsing process involves evaluating the low-level contract and translating it from its high-level representation into an intermediate transactional meaning executable by our Rust smart contract instance. In effect, transforming our contracts into an intermediate representation (IR) using previously defined language properties will produce a contract form that is faster to execute and process, allowing us to optimise the processing of contracts. Additionally, this intermediate contract form acts as a standardised representation for the end user.

The pseudocode presented in algorithm 6 simply loops through the contract string and updates its transactional variables, such as *amount* and *recipient*, on reading specific words in the string. This is a simpler process than previously described algorithms, as we are only working with low-level, non-compositional contracts. The created Contract object in line 41 of the pseudocode contains all necessary information required for the transaction and is used to display the contract in the web application, as well as to execute its contained transactions upon acquirement.

Algorithm 6 Transforming a Contract into Its Intermediate Representation

```
1:  procedure TRANSFORM(S)
2:      giveOccurrences ← 0
3:      getOccurrences ← 0
4:      getHasAppeared ← false
5:      amount ← 1
6:      contractObsArr ← []
7:      if S.includes(zero) then
8:          amount ← 0
9:      end if
10:     horizonDate ← GetHorizon(S)              ▷ update if 'truncate' is read
11:     for every word W ∈ S do
12:         if W = give then
13:             ++ giveOccurrences
14:         else if W = scaleK ∧ amount ≠ 0 then
15:             if next_word is int then
16:                 amount ← amount * next_word
17:                 ++ index
18:             else if next_word is observable then
19:                 contractObsArr.push(next_word)
20:                 ++ index
21:             end if
22:         else if W = get then
23:             getHasAppeared ← true
```

```
24:             ++ getOccurrences
25:         else if W = truncate then
26:             horizonDate ← next_word
27:             getHasAppeared ← false
28:         end if
29:     end for
30:     if getHasAppeared then
31:         return error                    ▷ get must be followed by truncate
32:     end if
33:     giveMod ← giveOccurrences mod 2
34:     if giveMod = 0 then recipient ← 0
35:     else recipient ← 1
36:     end if
37:     getMod ← getOccurrences mod 2
38:     if getMod = 0 then acquireAtHorizon ← false
39:     else acquireAtHorizon ← true
40:     end if
41:     Contract c ← new Contract(amount, contractObsArr, recipient,
42:                 horizonDate, acquireAtHorizon)
43: end procedure
```

To demonstrate an entire simulation of how a contract is processed step-by-step, a concrete example is given in the following figure.

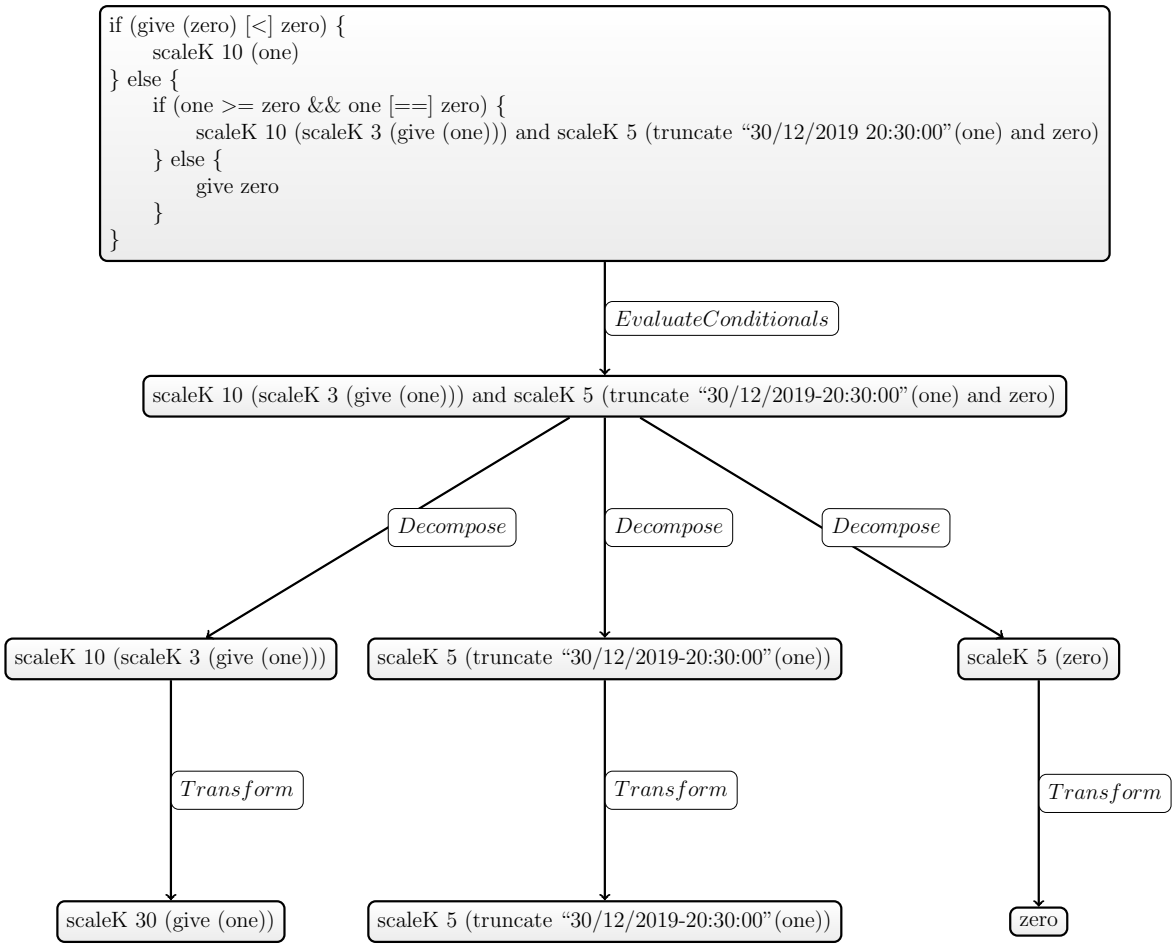


Figure 6.9: Contract Processing Simulation

6.2.5 Rust Code Execution

The pre-defined Rust Smart contract code that is used to execute blockchain transactions from the JavaScript Contract objects generated in section 6.2.4 is presented in Appendix A, where testing and event definitions have been left out for the sake of simplicity. We make use of a single smart contract definition that can be used for the execution of any arbitrary Nexus contract. As illustrated in the source code, our Rust smart contract definition is minimal, containing only functions that are required for two parties to interact with each other via transactions on the Ethereum blockchain. This enhances compatibility and maintainability, since functionality can be easily added and the system can be ported to any underlying blockchain. This will be demonstrated in the following section. By relying on a minimal Rust implementation, we are also preserving the security of the system by limiting the blockchain attack vector, because

essentially, less code goes into the smart contract.

6.2.6 Move IR Code Creation

Alongside the JavaScript contract object that is processed by our Rust smart contract instance upon acquirement, we also generate Move IR code from the intermediate representation of our Nexus code generated in section 6.2.4. Similarly to our pre-defined Rust smart contract definition, Libra transactions generated from Nexus input are limited to a few lines of code. The Move IR code that would be generated when creating a **one** contract, where the owner address is 0x004ec07d2329997267ec62b4166639513386f32e and the counter-party address is 0x7f023262356b002a4b7deb7ce057eb8b1aabb427, is illustrated in the following figure.

```
//! no-execute
import 0x0.LibraAccount;
import 0x0.LibraCoin;

main(payee: address) {
      let coin: R#LibraCoin.T;
      let account_exists: bool;
      let recipient: address;
      let sender: address;
      sender = 0x7f023262356b002a4b7deb7ce057eb8b1aabb427;
      recipient = 0x004ec07d2329997267ec62b4166639513386f32e;
      coin = LibraAccount.withdraw_from_sender(1);
      account_exists = LibraAccount.exists(copy(recipient));
      if (!move(account_exists)) {
            create_account(copy(recipient));
      }
      LibraAccount.deposit(move(recipient), move(coin));
      return;
}
```

Listing 6.1: Move IR Code Creation for 'one' Contract

This code is automatically downloaded through the browser onto the user's machine, which can then be locally run and tested for correctness via the Move command-line interface. There are two ways of testing the generated Move code. Firstly, one can test the correctness and semantics of the generated script via Libra's functional_tests testsuite. This allows users to exercise generated modules that modify the global blockchain state just like on a real blockchain. Secondly, one can run a local instance of the Libra blockchain on their machine and send transactions that publish the module, run the transaction script and so on, allowing users to run custom modules and

scripts in the libra_swarm command-line interface. More detail on this is mentioned in Appendix C. As of now, the documentation for this is sparse and the Libra Association have announced that they will be making this official in the following weeks. The original goal of this project was to be able to execute Nexus contracts on Ethereum, as well as the Libra blockchain, allowing for comparison and benchmarking between a permissionless and a permissioned implementation. However, although having opened a GitHub issue regarding our requirements [60], this is currently not possible, since Libra does not allow you to spawn a node locally as of yet. This is further discussed in section 9.2.4.

In effect, we have successfully implemented a source language for Move IR smart contracts using a high-level domain-specific language based on the Peyton Jones et al.'s previous efforts. This proves that our system can indeed be ported to any underlying blockchain at ease, whether permissioned or permissionless, while providing end users with a consistent high-level language and preventing end users from implementing any underlying smart contract code themselves.

Chapter 7

Testing

Testing is a necessary ingredient of any software development process, as it detects coding bugs early on throughout the development and verifies the behaviour of all system components in terms of their correctness and expectation. Furthermore, software testing ensures high code coverage, which is the degree to which our source code is executed when our system is used, limiting source code redundancy and generally improving the quality of individual system components. In essence, testing prevents unexpected failure of any system. This chapter will discuss the approaches that were followed for testing different components of our system.

Fundamentally, our system was tested iteratively. This agile development approach implies that individual components are tested at the end of each sprint, which refers to every cycle of development. Therefore, whenever new code was added to our software, its functionality and correctness was thoroughly tested before being merged with the main codebase. This ensured that newly added components did not break the functioning of existing parts. Our set of tests comprises both black-box tests, which test the functionality of individual units, as well as white-box tests, which test the internal structure of the system.

7.1 Unit Testing

Unit testing verifies the behaviour of individual units of a software, points out bugs early on during development and allows developers to avoid these at later stages of the development process. In essence, this provides a fast feedback loop that verifies the correctness of new components. When unit testing, the exact location of these errors is detected, generally making it easier to fix existing bugs while allowing developers to separately maintain each component. In addition to the above, developers gain a deeper understanding of the system from this type of testing, as the isolated nature of these tests often reveals extreme cases that developers do not anticipate. Moreover, unit tests act as a descriptive source of documentation, allowing developers to gain a fundamental understanding of individual lower-level units of the system. Unit testing facilitates compatibility and maintainability, as it requires units to be isolated, and generally creates a more structured codebase, making it easier to change or port existing

code.

Unit tests that were produced throughout the development of this project can be divided into JavaScript and Rust tests. Our collection of JavaScript tests entails 330 Unit tests that are run using the Mocha and Chai libraries provided by JavaScript. These tests are isolated from the source code and located in a separate testing folder, where each JavaScript file maintains its own testing file. JavaScript functions contained in the source code are run on extreme and erroneous inputs and the produced output is compared to the expected output using the *expect* and *assert* functions that Chai offers. This effectively ensures the correctness of our algorithms. A set of JavaScript tests are listed below.

```
describe('testing index.mjs...', function() {

    describe('decompose()', function() {
        // Output format: [first part, second part,
        // most balanced conj, stringToAddToBeginning,
        // stringToAddToEnd]

        context('with a non-empty contract string as argument',
    function() {

            it('1', function() {
                var res = decompose("one and ( zero or one )"
                  .split(" "));
                assert.isArray(res);
                assert.lengthOf(res, 5);
                assert.sameMembers(res, ["one", "( zero or one )",
                  "and", "", ""]);
            })

            it('2', function() {
                var res = decompose("give ( one and zero )"
                  .split(" "));
                assert.isArray(res);
                assert.lengthOf(res, 5);
                assert.sameMembers(res, ["give ( one )",
                  "give ( zero )", "and", "", ""]);
            })

            it('3', function() {
                var res = decompose("( give ( zero ) or
                  give ( one ) ) and ( one or zero )".split(" "));
                assert.isArray(res);
                assert.lengthOf(res, 5);
                assert.sameMembers(res, ["give ( zero ) or
                  give ( one )", "one or zero", "and", "", ""]);
            })
        })
```

```
    })
})
```

Listing 7.1: JavaScript Unit Testing

Additionally, a number of Rust unit tests were implemented inside our Rust contract definition to ensure the security of our smart contracts. For instance, these tests involve the testing of updates in balances that are expected after Ether is transferred, or the failure of a transfer transaction in the case where the sending address does not hold sufficient funds. An example of one of the implemented Rust tests is demonstrated below.

```
#[test]
fn transfer_and_update() {

    let mut contract = smart_contract::SmartContractInstance{};
    // initialise both addresses
    let holder = Address::from([0, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
   12, 13, 14, 15, 16, 17, 18, 19, 20]);

    let counterParty = Address::from([1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
    11, 12, 13, 14, 15, 16, 17, 18, 19, 20]);

    contract.constructor(holder, counterParty);
    // both balances should be 0
    assert_eq!(contract.holderBalance(), 0.into());
    assert_eq!(contract.counterPartyBalance(), 0.into());

    // Here we're creating an External context using ExternalBuilder
    // and set the `sender` to the `holder` address
    ext_update(|e| e.sender(holder.clone()));
    contract.depositCollateral(300.into());
    // holderBalance should be 300
    assert_eq!(contract.holderBalance(), 300.into());
    assert_eq!(contract.counterPartyBalance(), 0.into());

    contract.transfer(holder, counterParty, 200.into());
    // both balances should be updated
    assert_eq!(contract.holderBalance(), 100.into());
    assert_eq!(contract.counterPartyBalance(), 200.into());
}
```

Listing 7.2: Rust Unit Testing

7.2 Functional Testing

Functional tests involve tests that cannot be conducted in unit testing. These are likely to reveal bugs that are harder to detect, such as defects related to the integration of different system components. Functional testing tests the behaviour and functioning of the entire system, rather than some lower-level unit of the system. The following functional tests were carried out in order to test whether the functional requirements previously mentioned in chapter 4 are met by our system.

F1 The system shall allow users to easily construct smart contracts via a user-friendly interface of the web app;

A minimalistic web app interface is used that allows users to construct smart contracts via a single click.

F2 The system shall allow user input to be read, parsed, and used to create a smart contract instance;

The user input is efficiently parsed by our algorithms and made executable by our Rust contract.

F3 The system shall allow created contract code to be deployable on a blockchain via some blockchain client;

Parity is used to deploy our smart contracts on Ethereum.

F4 The system shall allow users to select a contract holder and contract counter party address for the deployment of a contract;

Input text fields are provided for the user to enter necessary addresses.

F5 The system shall allow users to select an oracle address that will provide any observable values when a contract is acquired;

Another text input field is presented to allow users to provide an address that will be the oracle for future contracts.

F6 The system shall allow users to deposit an arbitrary amount of Ether into both parties' accounts;

A select item is used to specify an arbitrary (integer) amount of Ether to be deposited into any of the accounts.

F7 The system shall allow users to add their own combinator definitions to extend the existing language;

A text input area can be used to provide custom-defined combinator definitions.

F8 The system shall visualise composed contracts in a table;

Composed contracts are presented in a table.

F9 The system shall present users with the option to choose a subcontract whenever a disjunction contract is provided;

Buttons for contract choices are presented during the parsing process.

F10 The system shall present the user with error logs in case a provided contract is syntactically incorrect;

Syntax checking of provided contracts prior to their execution notifies users in the case of any incorrect constructs.

F11 The system shall allow users to acquire non-expired supercontracts from the table of pending contracts;

A button labelled 'acquire' can be used to acquire pending supercontracts.

F12 The system shall allow users to calculate the value of pending supercontracts for any given time in the future;

A date select item can be used to calculate the future value (in Ether) of any pending supercontract for any given time.

F13 The system shall allow users to estimate the gas costs of composed contracts prior to execution;

This functionality has not been implemented in our system.

F14 The system shall allow expired contracts to become unavailable;

Acquired contracts are removed from the list of pending contracts. Once a supercontract in the table contains only acquired or expired contracts, its 'acquire' button becomes disabled.

7.3 Non-functional Testing

Non-functional tests involve tests that cannot be conducted in unit testing. These specify criteria that judge the operation and performance of a system. The following non-functional tests were carried out in order to test whether the non-functional requirements previously mentioned in chapter 4 are met by our system. The following non-functional tests were each carried out 10 times prior to calculating their average in order to obtain reliable results. 'Real-time' refers to the short period of time between the occurrence of an event and the triggered system response. Real-time responses should be in the order of milliseconds.

NF1 Contract transactions should be processed and sent to the blockchain network in real-time;

The average time taken for transactions to be processed and distributed to the blockchain network can be calculated as the sum of the time taken to process a user-defined contract plus the time taken for the contract to be distributed among the blockchain network after the user has pressed the 'acquire' button. On average, this time amounts to 0.92 seconds.

NF2 Contract transactions should be confirmed in real-time;

In general, this depends on the gas provided by the transactor, the local network, as well as the blockchain network's latency. Thanks to Parity's development blockchain network, the average time taken for contract transactions to be mined and confirmed was calculated to be as little as 0.35 seconds.

NF3 Expired contracts should become unavailable and labelled as 'expired' in less than 30 seconds after their time of expiry;

In order not to sacrifice contract processing performance, when removing recently expired contracts our system only iterates over the list of currently pending contracts in intervals of 15 seconds. Therefore, the maximum time taken for a contract expiry to be noticed should be 15 seconds (without interface update delays). The average time taken to update the user interface in this scenario was calculated to be 0.23 seconds, resulting in an overall average time of 15.23 seconds.

7.4 User Interface Testing

In addition to previously mentioned tests the web application was not only tested manually by the author, but was also distributed among a selected set of external participants with different technical ability. After having been introduced to Nexus and the use of our web application, the participants were allowed to play around with the web application interface for 5 minutes, before filling out a survey that documents the usability of our Nexus web application. 10 individuals chose to participate and the average results of the survey are presented in the figure below. The outcome of the survey was promising.

Comprehensibility	**Functionality**	**Design**
8.2/10	9.8/10	8.4/10

Figure 7.1: UI Usability Survey Results

7.5 Remarks

To conclude, a wide range of tests were conducted in order to allow for a successful development of this project, as well as to verify the behaviour of different components of our system. The results of these tests were promising and will be evaluated in the following chapter.

Chapter 8

Evaluation

This chapter will evaluate the product of this project with respect to performance and usability. We will evaluate its real-word application and compare Nexus to existing systems. At the end of this chapter, we will evaluate whether the requirements and specifications mentioned in section 4 have been met.

8.1 Evaluation Against Requirements

We have successfully implemented a domain-specific language for financial smart contracts. Our system implementation processes simple user input composed in our language and uses this to create smart contracts on the Ethereum blockchain. All functional requirements of the system have been met apart from F13. As of now, users are not able to estimate the gas costs of their custom-defined smart contracts, because the time limit imposed on this project did not allow for the implementation of this functionality. We believe that the implementation of this feature is non-trivial and should be included in future versions of our system. All in all, the functionality necessary to satisfy our initial goals for this project is present.

In terms of usability, the survey results listed in figure 7.1 suggest that the average end user, no matter how much technical knowledge they possess, is more than satisfied with our web application user interface. A particularly strong score of 9.8 was noticed for the functionality offered by our web application. The relatively lower score of 8.2 present for the comprehensibility of the user interface is likely due to the amount of technical terms used. This is due to the fact that, generally speaking, the blockchain industry is surrounded by a lot of technical jargon.

Our system software relies on a small number of external libraries. This has been proven to be very beneficial, as our implementation could easily be ported from Rust to Move IR smart contracts. This proves the presence of a considerable degree of compatibility in our system.

Our final system possesses high performance, especially regarding the smart contract interaction over our pre-defined Rust contract implementation. Having a trivial Rust

smart contract definition that the user does not directly interact with, allows for fast transaction processing, as the bulk of the work, that being the parsing and compilation of the Nexus contract input, is done in the background of the web application. In the blockchain world performance is an absolute key factor for usability, which is why background compilation and contract deployment should not introduce any time lags. As demonstrated in the previous chapter, all non-functional requirements have been met. Additionally, error messages are displayed in our web application that notify the end-user when something goes wrong or an action time out occurs.

Last but not least, during the iterative development of this project, our system has proven to be maintainable and reusable. Modifications can be added to the system in one place, without affecting the functioning of the remaining application, furthering the vision of Nexus as an open-source project. This has been achieved through extensive testing throughout the development process together with a coherent architectural code structure.

8.2 Contract Processing Performance

This section will evaluate the performance of our JavaScript contract processing algorithm, namely *ProcessContract*, in terms of its execution time. Firstly, we will test the performance of the underlying *EvaluateConditionals* function, which is one of the more expensive parts of the contract processing process. Throughout this section, contract string inputs of increasing size will be constructed and used as function input to be tested. We will run the algorithms on 201 inputs of increasing size, starting from 0 (a non-nested contract string) up to 200 (a nested contract string made up of 200 contracts). This will effectively provide us with enough data to deduce a conclusion about our algorithm's performance. For each input we will repeat our tests 10 times before calculating and plotting the average result. The test results will be plotted in intervals of 5 input steps.

8.2.1 EvaluateConditionals

The *EvaluateConditionals* function is responsible for evaluating conditional statements, retrieving the corresponding consequence (or alternative in cases where the condition evaluates to false) and appending this to the currently parsed contract string. The first test on our JavaScript implementation of this algorithm was conducted with **if (one** [>] **zero) {zero} else {one}** as the input contract string. The result of this was calculated to be 0.13 ms, which is plotted in figure 8.1 where input size equals 0. The input for every following plot simply replaces the **one** combinator inside the if-condition by the initial input. This is repeated for every new test, gradually increasing and adding complexity to the input size. Therefore, the input string for n=1,2,... will be **if (if (one**[>]**zero) {zero} else {one}** [>] **zero) {zero} else {one}**, **if (if (if (one** [>] **zero) {zero} else {one}** [>] **zero) {zero} else {one}** [>] **zero) {zero} else {one}**, and so on. While the results presented in figure 8.1 seem unconsistent for the first few tests, by connecting all the points in the graph we can spot a linear runtime. Overall our algorithm is highly efficient with a runtime of 0.16ms needed to evaluate a contract string consisting of 100 nested conditional statements.

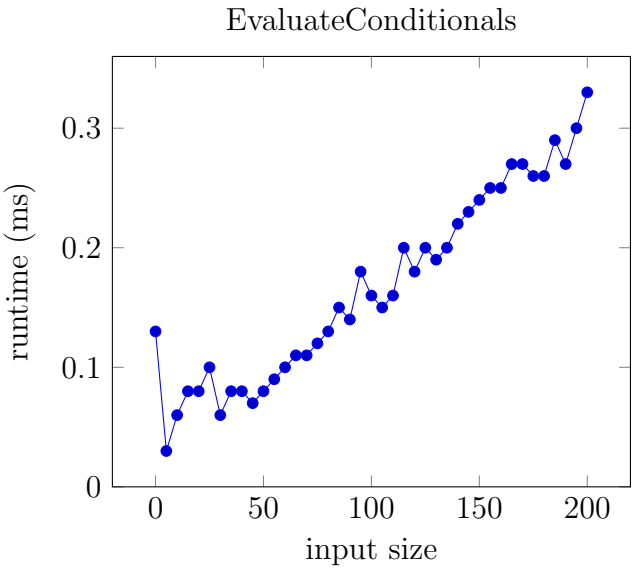


Figure 8.1: Performance of EvaluateConditionals

8.2.2 ProcessContract

Algorithm *ProcessContract* is the main contract processing algorithm, which makes use of the *EvaluateConditionals* algorithm (which in turn uses *GetHorizon* and *GetValue*), as well as the *Decompose* algorithm. Therefore, by testing the performance of this algorithm we can certainly deduce the performance of the entire contract processing step in our system. In order to test the *ProcessContract* function, we had to make slight modifications to the source code, because the definition of this function is extensive and handles user interface updates, which would otherwise terminate our performance tests prematurely. An example of such a user interface interaction is the display of disjunct contract choices to the end user. This is simply replaced by an additional recursive *ProcessContract* call on both highest-level subcontracts, allowing our algorithm to continue processing the contract string until no more connectives can be found.

As explained in section 6.2.2, the performance of the *Decompose* algorithm is proportional to the number of *or* connectives contained inside the input contract string. This is because our algorithm needs to pause to present the user with a disjunct contract choice, and eventually recurses on the chosen subcontract string. While enhancing usabiliy of the system, this adds additional overhead to our *ProcessContract* algorithm. The first test on our JavaScript implementation of this algorithm was conducted with **zero or one** as the input contract string. The result of this was calculated to be 1.11 ms, which is plotted in figure 8.2 where input size equals 0. The input for every following plot simply replaces the **one** combinator by **scaleK 5(zero and one)**, which is repeated for every new test, gradually increasing and adding complexity to

the input size. Therefore, the input string for n=1,2,... will be **zero and scaleK 5 (zero and one)**, **zero or scaleK 5 (zero and scaleK 5 (zero and one))**, and so on. The results plotted in figure 8.2 suggest a non-linear exponential complexity of our JavaScript algorithm implementation.

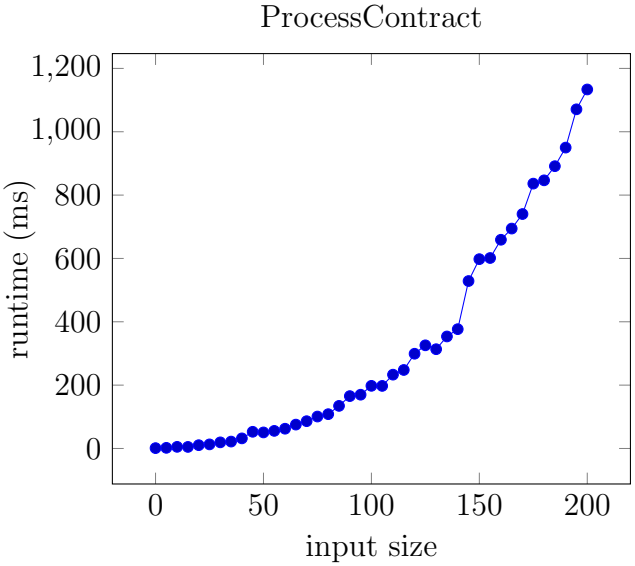


Figure 8.2: Performance of ProcessContract

Naturally, *ProcessContract* takes considerably more time to execute than the previously evaluated algorithm. However, in the tests above our input strings only ever contained a single *or* connective. This connective also turns out to be the outermost connective in every test case, and hence the first connective to be extracted. As a result, *Decompose* will only be called twice (the first time on the initial string and the second time on the right substring) and *DecomposeByAnds* will be called for the remainder of the function runtime. Therefore, our test results above do not accurately illustrate the worst-case scenario. In order to demonstrate this slowing effect caused by the *Decompose* function, we repeated the tests above, but instead of replacing the **one** combinator by **scaleK 5(zero and one)** we replaced it by **scaleK 5(zero or one)**, effectively creating compositional contracts consisting of *or* connectives only. The results of this revised test are plotted in the following figure.

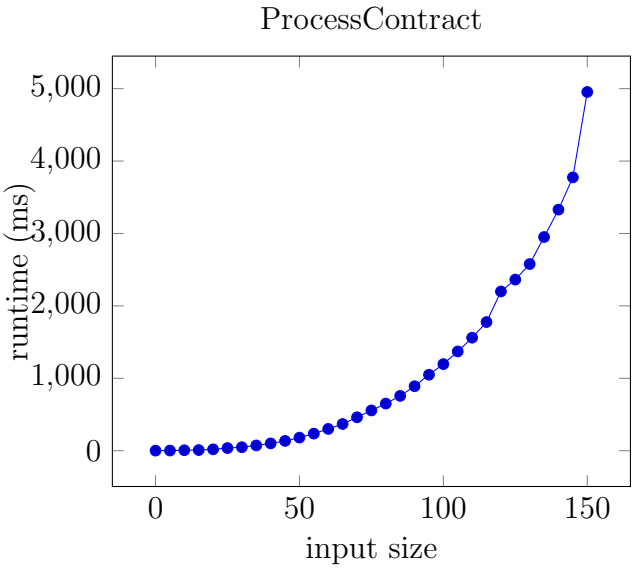


Figure 8.3: Performance of ProcessContract (revised)

While the two graphs in this subsection both start off with a similar gradient for the first 10 inputs, the revised tests slow down faster than the original set of tests. As a comparison, the original tests take 598 ms to process a contract of input size equal to 150, whereas the same algorithm takes about 4,954 ms to execute with the revised tests, demonstrating just how important it is to test algorithms on an extensive range of inputs. The graph plotted in the figure above accurately represents the graph of an exponential algorithm. This exponential runtime is caused by the fact that *Decompose* continuously recurses until no more connectives are present in the input string.

Although we did not cover all possible combinations of test cases in this section, it can be concluded that the exponential runtime of our contract processing algorithm will become very expensive for smart contracts of input size greater than 150, that being contracts consisting of over 150 nested subcontracts. Such contract may take about 5 seconds to process, possibly resulting in time lags in the user interface of the application. In addition to the above, our inputs in this section did not contain any conditional statements. This would add the runtime from the previous section on top of the runtimes present in this section for any point in the graph. Furthermore, our tests neglect the process of replacing custom-defined user definitions, string modifications such as the formatting of date, as well as syntax checking of input contracts, which will each add additional overhead to the runtime of our contract processing algorithm. However, this can be assumed to be minimal as compared to the runtime of the *EvaluateConditionals* algorithm. Moreover, it is worthy to mention that our system is aimed at simple smart contracts and at users with little technical background, thus, unlikely to require the handling of inputs of such scale.

8.3 Use Cases & Gas Costs

This section will evaluate how the system behaves when applied to different use cases. We evaluate the ease of implementation as well as the costs (in terms of gas) of Nexus contracts and benchmark this with equivalent smart contracts implemented in Solidity code.

In order to compare our system's costs in terms of gas consumed by Nexus smart contracts, we implemented a set of smart contracts in Solidity [61] that involves all combinators of our language. Together with our Rust smart contract instance implementation, these Solidity implementations can be found in the Appendix. In order to run these benchmarking tests, the web application was launched, the tested set of contracts were acquired, and their gas costs were retrieved from MetaMask. To ensure that our tests are reliable, we used different development environments, and each test was repeated 10 times before averageing the gas cost output.

First of all, we tested the gas consumption of the probably simplest smart contract in our domain-specific language: **one**. An equivalent Solidity implementation was implemented by Dean [61] and is listed in Appendix B.1. Because our system separates the composing of high-level Nexus contract code from its transformation into Rust code in the background, the input required to execute this contract in our system simply corresponds to **one**, since the Rust smart contract instance has been pre-defined by the author. Secondly, we tested the gas consumed by a so-called Zero-Coupon Discount Bond (ZCB). The ZCB is a commonly known accrual bond in the financial industry and in Nexus, it can be represented as **scaleK k (get (truncate t (one)))**. This smart contract pays k Ether to the contract owner at time t (and no earlier or later than t). Again, similarly to our previous Solidity implementation, our single line Nexus code complexity cannot be compared to the amount of code that needs to be developed by the end user for the Solidity version listed in Appendix B.2. Lastly, we tested the gas consumption of a European Option (EO) contract. A European Option financial contract allows the contract owner to exercise choices between underlying contracts. More specifically, a European Option gives the contract owner the right to choose, at a particular date, whether or not to acquire an underlying contract [1]. Therefore, a European Option contract can be represented in Nexus using a combination of the *get* and *truncate* combinators, as well as the *or* connective: **get (truncate t (c or zero))**. This contract implies that at time t, the owner of the contract can choose whether or not to acquire underlying contract c. Unsurprisingly, the Solidity code required to implement our European Option example listed in Appendix B.3 is more complex than its equivalent Nexus code consisting of 5 combinators only.

The three Nexus contracts that were tested against their Solidity counterparts include **one**, **scaleK 100 (get (truncate "01/01/2020 12:00:00" (one)))** and **get (trun-**

cate "01/01/2020 12:00:00" (one or zero)). The two figures below present the results of our gas consumption tests. For reference, the gas consumed by an Ethereum transaction corresponds to the fee that the transactor is required to pay to execute the transaction. In this section gas is given in Wei and 10^{18} Wei corresponds to 1 Ether, which equals £143.86 at the time of writing. Figure 8.4 compares the gas consumption in Wei of deploying the mentioned Nexus contracts with the deployment of their equivalent Solidity implementation. In this test, the traditional Solidity contract deployment clearly beats our Rust contract deployment. The figure shows that the cost of deploying our Rust smart contract is more than twice the cost of deploying any of the tested contracts in Solidity, where the gas consumed by our system amounts to 3,440,000 Wei and the average gas consumed in Solidity equals 1,210,000 Wei. This significant difference is likely caused by the number of pWasm library imports, as well as the number of functions declared in the Rust smart contract instance. Our Rust contract defines 11 contract functions, whereas the Solidity implementations declare at most 8 functions. These functions require storage on the Ethereum blockchain and thus result in higher gas fees. However, since the tests involved only rather simple smart contracts, because implementing highly complex compositional Nexus contracts in Solidity was beyond the scope of this project, it can be deduced that the outcome of our tests represented in the graph is only valid for trivial smart contracts. This can be stated with certainty, because as we test more complex contracts, our Rust definition does not vary nor increase in size, whereas the Solidity contract definition will drastically expand. Therefore, it can be stated that for larger compositional contracts our Rust code deployment cost remains constant and will certainly become cheaper than the deployment of a complex Solidity solution.

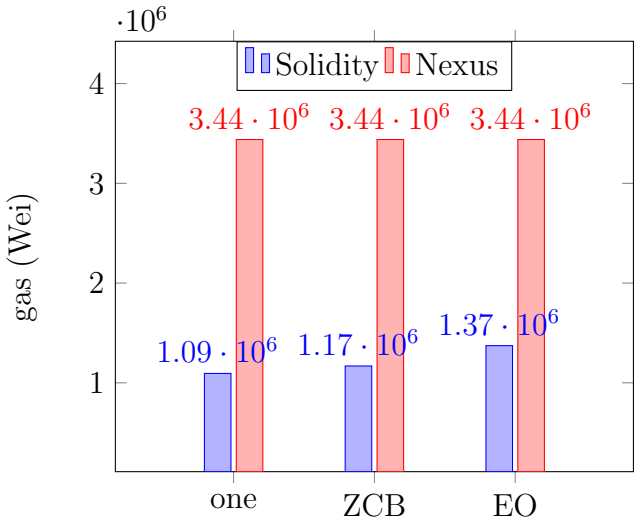


Figure 8.4: Constructor Call Gas Costs

Figure 8.5 shows the gas consumption in Wei of acquiring, that is executing the un-

derlying transactions of our three smart contract definitions. The data shown proves that Nexus can certainly compete with Solidity's transaction execution in terms of the costs needed to be spent by the transactor. For all three contracts, Nexus beats Solidity in terms of gas consumed. As stated previously, the existing difference between Solidity and Nexus gas costs of acquiring a contract is likely to increase as the tested contracts become more compositional, especially due to the *acquire* function in our Solidity implementations becoming more elaborate because of additionally introduced safety checks.

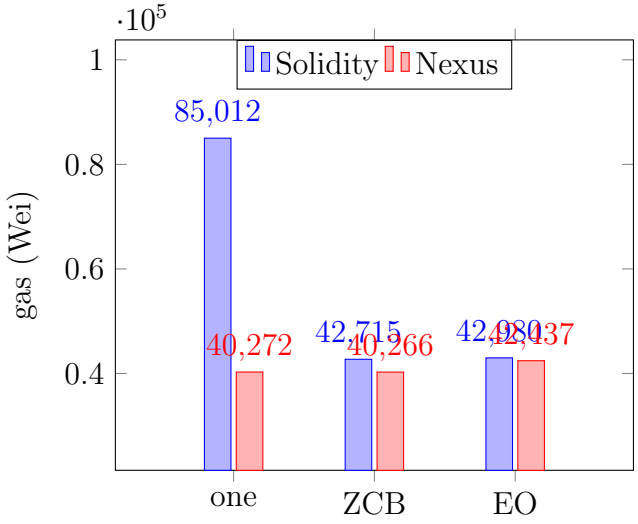


Figure 8.5: Acquire Call Gas Costs

Generally speaking, the Solidity code that needs to be developed by the end-user to execute the given set of tested contracts is non-trivial. The Solidity implementations given each contain over 150 lines of code, which is more than our Rust smart contract implementation alone. Firstly, this is caused by Solidity's underlying security vulnerabilities, which require developers to include numerous safety checks (of which some are enforced by the EVM) in order to write "safe" Solidity contracts. In the Solidity implementations given, such checks involve the *safeAddSigned* and *safeSubSigned* functions, which check for and prevent integer over- and underflows. Additionally, the *withdraw* function updates the balance before eventually transferring Ether. This order of execution is necessary in order to prevent a possible reentrancy attack, further demonstrating the risks associated with Solidity smart contract development. Secondly, our Solidity implementations make use of numerous modifiers, such as *onlyHolder*, which proceeds to the execution of the function body only if the function is called by the contract owner. These safety precautions are needed to ensure resilience against known Solidity attacks. In our system, on the other hand, the JavaScript web application code handles these, thus removing the need for such checks in the Rust code. Last but not least, when writing any Solidity smart contract all generic functions, for instance functions

handling the transfer or the deposit of Ether, need to be implemented from scratch and cannot be separated from the contract itself. Therefore, in our Solidity implementations several definitions, such as the *withdraw* and *stake* functions, are repeated. This is one of the main issues of Ethereum that requires developers to write extensive smart contracts, making Solidity smart contract development highly prone to coding errors. Nexus does not require users to compose complex smart contract code, but instead allows the end-user to use our simple high-level language instead, ensuring usability of the system and effectively reducing barriers to entry for users. With Nexus, the user can use the web application to provide inputs as simple as **one**, which will automatically be transformed and handled by the system in the background — **one** and the complexity of the code needed to produce an equivalent Solidity contract (see Appendix B.1) are beyond compare. Essentially, this diminishes any sort of coding error introduced by the end-user.

To conclude, smart contracts are undoubtedly easier to implement in Nexus than in Solidity, essentially resulting in Nexus contracts being less prone to errors and security threats. While the gas costs of deploying Nexus contracts is higher for trivial contracts, this cost remains constant as contracts become more complex, and the costs of deploying complex Solidity contracts will eventually overtake. However, gas costs may play an insignificant role depending on the use of Nexus. This is because nowadays financial institutions often run their own private blockchain, without incurring internal gas fees.

Chapter 9

Conclusion

This chapter will conclude our paper. We will describe the outcome of the project and how this compares to existing solutions, conveying the potential of Nexus. Following this, we will go on to introduce future ideas involving functionality that could be modified or added, as well as technical improvements that were simply beyond the scope of this time-bounded implementation. In effect, this section will describe the next steps in the vision of Nexus. Finally, we will reflect on the overall project outcome.

9.1 Achievement

It can be concluded that we have successfully managed to create a domain-specific language for financial smart contracts that allows users to construct Ethereum smart contracts in a simple manner. This was achieved by creating a language based on carefully-defined combinators taken from [1]. Nexus is accompanied by a web application, which is vital in order to isolate Nexus contracts defined by the end user from the pre-defined Rust smart contract implementation. Thanks to this isolation of high-level Nexus code from lower-level Rust smart contract code, our system is easily portable to any underlying blockchain. This is demonstrated when the compiled Nexus output is used to produce equivalent Move IR smart contract code, effectively using Nexus as source language for Move IR smart contracts. Moreover, the end-user is not required to develop any smart contract code themselves, because all functionality of the final smart contract instance is pre-defined and completely isolated from the user interface. This not only makes our system trivial to operate even for end users with little technical background, but also limits security vulnerabilities that are present with conventional smart contract programming languages. The simplicity and usability of our web application interface makes the underlying complexities of financial contracts and blockchain technology virtually invisible to the end user, ultimately allowing users to solely concentrate on the logic of the Nexus contracts they compose.

Our web application allows end-users to construct custom-defined contracts, add their own definitions to extend the language, as well as evaluate and manage contracts. While the evaluation model introduced by Peyton Jones et al. [1] is only abstract and was not implemented in a real-world financial setting, we built on this by enabling

users to evaluate pending smart contracts for any given time in the future. Ultimately, this allows our web application to help users understand more about the evolution of contracts during their life, but also learn more about general market activity. Furthermore, this offers a way of simulating and visualising different transaction orderings in the web application, something that many existing systems are lacking.

In terms of contract execution costs in gas, Nexus contracts have been proven to be cheaper than their equivalent Solidity counterparts. Nexus is efficient in the user input, and although the performance of our algorithms that process Nexus contracts becomes exponentially more expensive for highly nested compositional contracts, this overhead can be disregarded for most applications, as our system is not expected to be used on contracts composed of more than 100 subcontracts.

9.2 Next Steps & Future Ideas

9.2.1 Nexus Language

Future goals of our system involve the improvement of the Nexus language itself. This includes expanding the set of combinators that build our language, which involves implementing the *anytime* and *then* combinators introduced by Peyton Jones et al. [1], as well as additional combinators that will allow us to describe a wider range of financial contracts. The former would be trivial to add to our implementation.

Secondly, we would like to analyse our language in greater detail and evaluate it in an additional set of use cases, such as its application to an American Options contract or the use case described in the Obsidian paper published by Koronkevich [62]. We would also like to formulate additional mathematical proofs about our language and its semantics to allow us to add to our contract theorems that are mentioned in previous sections. Such proofs can be applied in order to optimise our evaluation process, as well as to enhance the compilation process into our intermediate contract representation.

Furthermore, as of now, consequences in conditional statements cannot contain conditionals themselves. This implies that in order to apply nested else-if-statements the user would have to nest *else* and *if* separately, adding additional indentation and nesting levels, thus slowing down performance and reducing the readability of compositional Nexus contracts. Instead Nexus should make use of an *else if* construct, which is commonly known in other programming languages such as Java. This should be implemented in future versions of Nexus.

Lastly, it would be interesting to implement interest as well as exchange rates. By adding these, for instance, the value of **get(c)** would become dependent on the interest rates present. Peyton Jones et al. [1] already suggest approaches to implement this using the Ho-Lee interest rate model [63] to estimate future interest rates. Fundamentally, this will use something referred to as lattice to estimate the range of what a value will be in the future.

9.2.2 Performance

Although the usability of our system should be kept as priority, there is definitely room for optimisations of our algorithms. This involves updating the algorithms so that syntax checking is done during the parsing process and the *EvaluateConditionals* and *Decompose* algorithms could also be merged, effectively only ever parsing the contract string once, unless there is a disjunct contract option to be displayed. Secondly, our system could be improved by relying on a payment channel between the two parties (see section 5.3). This will not only result in lower transaction fees for both parties involved, but also enhances the performance of our system.

9.2.3 Usability

In terms of usability, we could allow more than two parties to be involved in a contract. This would be non-trivial to implement, however, one could look at existing solutions, such as linked payment channels, to implement this. Additionally, allowing users to estimate the gas costs of their smart contracts would further enhance the usability of our system.

9.2.4 Libra Integration

The Libra documentation online is coherent and well explained. The papers that Libra has made available on their website [33, 38, 29] are neatly written and provide a great insight into the underlying blockchain and the concepts of the Move language. In addition to this, Libra provides a well kept "Getting Started With Move" guide, which allows users to play around with the language and also provides useful troubleshooting tips.

Ethereum is a permissionless blockchain. It would have been interesting to implement the system for the permissioned Libra blockchain and compare the two implementations in terms of usability, scalability and performance. However, because Libra is in its early

stages of development and Libra only provides the intermediate representation of the Move language, that being Move IR, this was not possible to do. As of now, users are only able to test the semantics of Move IR code. It is impossible to spawn and run a local Libra node that lets you run custom modules using a dummy global state with a few test accounts. We have contacted the Libra Association about our requirement, but we were informed that they are currently working on a testing environment that will allow for our requirements in the "near future". Our system does, however, produce Move IR transaction code, so that as soon as the user input is parsed and a contract is being executed, the contract is transformed into Rust as well as the equivalent Move IR source code, which is automatically downloaded onto the user's local machine to be tested and run as described in Appendix C.

Implementing Nexus on Libra will be one of the main objectives for future versions of this project. In order to be able to combine our existing system with the Libra blockchain, Wasm must allow Libra code to be compiled into Wasm bytecode, which can then be further compiled into JSON contract code executable on the Ethereum blockchain. This will require a Move source code script that offers the same functionality as our currently defined Rust smart contract implementation. Ultimately, this would allow our system to use the two blockchains interchangeably.

9.3 Reflection

To conclude, this project was highly successful. We successfully managed to achieve the goals that were set at the early stages of the development process. Using blockchain technology for financial contracts is a hot topic in the financial and technology industry, and certainly brings substantial benefits. The proposed system shows great potential and although the time bounds imposed on this project limited the scope of our implementation, the system should certainly be extended for future use in order to allow the development of smart contracts to be simpler and safer. In this paper we presented Nexus: a cutting-edge technology that allows users to compose financial smart contracts securely, with ease and at little cost. The advantages of using Nexus instead of existing smart contract programming languages are undeniable. We have effectively created a domain-specific language for writing safer smart contracts.

Appendices

Appendix A

Rust Smart Contract

```
#![no_std]
#![allow(non_snake_case)]
#![feature(proc_macro_hygiene)]

extern crate pwasm_std;
extern crate pwasm_ethereum;
extern crate pwasm_abi;
extern crate pwasm_abi_derive;

// Declares the dispatch and dispatch_ctor methods
use pwasm_abi::eth::EndpointInterface;
use pwasm_ethereum::{ret, input};

#[no_mangle]
pub fn deploy() {
     let mut endpoint = smart_contract::SmartContractEndpoint::new(
          smart_contract::SmartContractInstance{});
     endpoint.dispatch_ctor(&input());
}

pub mod smart_contract {
     use pwasm_std::types::{H256, Address, U256};
     use pwasm_ethereum::{read, write, sender};
     use pwasm_abi_derive::eth_abi;

     fn holder_key() -> H256 {
          H256::from
    ([0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0])

     }

     fn counter_party_key() -> H256 {
          H256::from
    ([1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0])

     }

     #[eth_abi(SmartContractEndpoint, SmartContractClient)]
     pub trait SmartContract {
          /// The constructor
```

```
37          fn constructor(&mut self, holder_address: Address,
38               counter_party_address: Address);
39          #[constant]
40          fn balanceOfAddress(&mut self, _address: Address) -> U256;
41          #[constant]
42          fn holderBalance(&mut self) -> U256;
43          #[constant]
44          fn counterPartyBalance(&mut self) -> U256;
45          #[constant]
46          fn holderAddress(&mut self) -> H256;
47          #[constant]
48          fn counterPartyAddress(&mut self) -> H256;
49          fn transfer(&mut self, _from: Address, _to: Address,
50            _amount: U256);
51          #[payable]
52          fn depositCollateral(&mut self, amount: U256);
53      }
54
55      pub struct SmartContractInstance;
56
57      impl SmartContract for SmartContractInstance {
58          fn constructor(&mut self, holder_address: Address,
59            counter_party_address: Address) {
60              write(&holder_key(),
61                &H256::from(holder_address).into());
62              write(&counter_party_key(),
63                &H256::from(counter_party_address).into());
64          }
65
66          fn balanceOfAddress(&mut self, address: Address) -> U256 {
67              read(&balance_key(&address)).into()
68          }
69
70          fn holderBalance(&mut self) -> U256 {
71              read_balance(&address_of(&holder_key())).into()
72          }
73
74          fn counterPartyBalance(&mut self) -> U256 {
75              read_balance(&address_of(&counter_party_key())).into()
76          }
77
78          fn holderAddress(&mut self) -> H256 {
79              read(&holder_key()).into()
80          }
81
82          fn counterPartyAddress(&mut self) -> H256 {
83              read(&counter_party_key()).into()
84          }
85
86          fn transfer(&mut self, from: Address, to: Address,
87            amount: U256) {
```

```
            let senderBalance = read_balance(&from);
            let recipientBalance = read_balance(&to);

            if senderBalance < amount {
               } else if to == from {
               } else if amount == 0.into() {
               } else {
                    let new_sender_balance = senderBalance - amount;
                let new_recipient_balance = recipientBalance
                      + amount;
                write(&balance_key(&from),
                      &new_sender_balance.into());
                write(&balance_key(&to),
                      &new_recipient_balance.into());
            }
        }

          fn depositCollateral(&mut self, amount: U256) {
               let sender = sender();
               let sender_balance = read_balance(&sender);
            let new_sender_balance = sender_balance
                 + amount;
               write(&balance_key(&sender),
                 &new_sender_balance.into());
          }

    }

    fn address_of(key: &H256) -> Address {
         let h: H256 = read(key).into();
         Address::from(h)
    }

    fn read_balance(owner: &Address) -> U256 {
         read(&balance_key(owner)).into()
    }

    // Generates a balance key for some address.
    // Used to map balances with their owners.
    fn balance_key(address: &Address) -> H256 {
         let mut key = H256::from(*address);
         key.as_bytes_mut()[0] = 1;
         key
    }
}
```

Listing A.1: Rust Smart Contract Implementation

Appendix B

Solidity Implementations

B.1 Simple Contract

```
pragma solidity >=0.4.22 <0.6.0;

contract One {
    // Static values
    int256 MAX_INT256 = int256(~(uint256(1) << 255));
    int256 MIN_INT256 = int256(uint256(1) << 255);

    // The contract holder
    address holder;

    // The counter-party
    address counterParty;

    // The stakes of the holder and counter-party
    mapping(address => int256) stakes;

    // Whether or not this contract has been acquired
    bool acquired;

    // Constructor, takes the contract holder address
    constructor(address contractHolder) public {
        require(
            contractHolder != msg.sender,
            "Holder and counter-party cannot have the same address."
        );
        // Set the holder and counter-party
        holder = contractHolder;
        counterParty = msg.sender;

        // Initialise stakes to 0 and acquired to false
        stakes[counterParty] = 0;
        stakes[holder] = 0;
        acquired = false;
    }

    // Only allows the holder or counter-party to call a function
```

```
37      modifier onlyParties() {
38          require(
39              msg.sender == counterParty || msg.sender == holder,
40              "Can only be called by holder or counter-party."
41          );
42
43          _;
44      }
45
46      // Only allows the holder to call a function
47      modifier onlyHolder() {
48          require(
49              msg.sender == holder,
50              "Only the holder may call this function."
51          );
52
53          _;
54      }
55
56      // Returns the balance of one of the two parties
57      function getBalance(bool holderBalance) public view returns
58        (int256) {
59          if (holderBalance) {
60              return stakes[holder];
61          } else {
62              return stakes[counterParty];
63          }
64      }
65
66      // Acquires this contract
67      function acquire() public onlyHolder() {
68          require(
69              !acquired,
70              "This function can only be called before acquisition."
71          );
72
73          acquired = true;
74
75          // Update balances
76          transferToHolder(1);
77      }
78
79      // Stake Ether in the contract
80      function stake() public payable onlyParties() {
81          require(
82              uint256(MAX_INT256) >= msg.value,
83              "Value staked is too big to be stored as int256."
84          );
85
86          // Update balance
87          stakes[msg.sender] = safeAddSigned(stakes[msg.sender],
```

```
88            int256(msg.value));
89    }
90
91    // Withdraw Ether from the contract
92    function withdraw(uint64 amount) public onlyParties() {
93        require(
94            address(this).balance > 0,
95            "Contract does not have enough funds."
96        );
97        require(
98            stakes[msg.sender] > 0,
99            "The caller does not have enough stake."
100       );
101
102       uint64 finalAmount = amount;
103
104       // Clamp withdrawal amount to total contract balance
105       if (address(this).balance < finalAmount) {
106           finalAmount = uint64(address(this).balance);
107       }
108
109       // Clamp withdrawal amount to party's balance
110       if (stakes[msg.sender] < finalAmount) {
111           finalAmount = uint64(stakes[msg.sender]);
112       }
113
114       // Adjust balance first to prevent re-entrancy bugs
115       stakes[msg.sender] = safeSubSigned(stakes[msg.sender],
116         int256(finalAmount));
117
118       // Send Ether (with no gas)
119       msg.sender.call.value(finalAmount).gas(0);
120   }
121
122   // Transfers the given amount from the holder to the counter-
      party
123   function transferToHolder(int256 amount) private {
124       stakes[holder] = safeAddSigned(stakes[holder], amount);
125       stakes[counterParty] = safeSubSigned(stakes[counterParty],
126         amount);
127   }
128
129   // Add two signed integers if no overflow or underflow can occur
130   function safeAddSigned(int256 a, int256 b) private view
131     returns (int256) {
132       require(
133           (b >= 0 && a <= MAX_INT256 - b) ||
134           (b < 0 && a >= MIN_INT256 - b),
135           "Integer overflow or underflow."
136       );
137
```

```
138         return a + b;
139     }
140 
141     // Subtract one signed integer from another if no overflow or
    underflow can occur
142     function safeSubSigned(int256 a, int256 b) private view
143       returns (int256) {
144         require(
145             b != MIN_INT256,
146             "Integer overflow or underflow."
147         );
148 
149         return safeAddSigned(a, -b);
150     }
151 }
```

Listing B.1: Solidity Implementation of 'one' [61]

B.2 Zero-Coupon Discount Bond

```
pragma solidity >=0.4.22 <0.6.0;

contract EuroOption {

    // Static values
    int256 MAX_INT256 = int256(~(uint256(1) << 255));
    int256 MIN_INT256 = int256(uint256(1) << 255);
    uint256 HORIZON_UNIX = 1577880000;

    // The contract holder
    address holder;

    // The counter-party
    address counterParty;

    // The stakes of the holder and counter-party
    mapping(address => int256) stakes;

    // Whether or not this contract has been acquired
    bool acquired;

    // Constructor, takes the contract holder address
    constructor(address contractHolder) public {
        require(
            contractHolder != msg.sender,
            "Holder and counter-party cannot have the same address."
        );
        // Set the holder and counter-party
        holder = contractHolder;
        counterParty = msg.sender;

        // Initialise stakes to 0
        stakes[counterParty] = 0;
        stakes[holder] = 0;
        acquired = false;
    }

    // Only allows the holder or counter-party to call a function
    modifier onlyParties() {
        require(
            msg.sender == counterParty || msg.sender == holder,
            "can only be called by the holder or counter-party."
        );

        _;
    }

```

```
48      // Only allows the holder to call a function
49      modifier onlyHolder() {
50          require(
51              msg.sender == holder,
52              "Only the holder may call this function."
53          );
54
55          _;
56      }
57
58      // Returns the balance of one of the two parties
59      function getBalance(bool holderBalance) public view
60        returns (int256) {
61          if (holderBalance) {
62              return stakes[holder];
63          } else {
64              return stakes[counterParty];
65          }
66      }
67
68      // Acquires this contract
69      function acquire() public onlyHolder() {
70          require(
71              !acquired,
72              "This function can only be called before
73                acquisition."
74          );
75          require(
76              now <= HORIZON_UNIX,
77              "This contract can only be acquired until
78                01/01/2020 12:00:00 UTC."
79          );
80
81          acquired = true;
82
83          if (now < HORIZON_UNIX) {
84              return;
85          }
86
87          // Update balances
88          transferToHolder(100);
89      }
90
91      // Stake Ether in the contract
92      function stake() public payable onlyParties() {
93          require(
94              uint256(MAX_INT256) >= msg.value,
95              "Value staked is too big to be stored as int256."
96          );
97
98          // Update balance
```

```
99          stakes[msg.sender] = safeAddSigned(stakes[msg.sender],
100           int256(msg.value));
101     }
102
103     // Withdraw Ether from the contract
104     function withdraw(uint64 amount) public onlyParties() {
105         require(
106             address(this).balance > 0,
107             "Contract does not have enough funds."
108         );
109         require(
110             stakes[msg.sender] > 0,
111             "The caller does not have enough stake."
112         );
113
114         uint64 finalAmount = amount;
115
116         // Clamp withdrawal amount to total contract balance
117         if (address(this).balance < finalAmount) {
118             finalAmount = uint64(address(this).balance);
119         }
120
121         // Clamp withdrawal amount to party's balance
122         if (stakes[msg.sender] < finalAmount) {
123             finalAmount = uint64(stakes[msg.sender]);
124         }
125
126         // Adjust balance first to prevent re-entrancy bugs
127         stakes[msg.sender] = safeSubSigned(stakes[msg.sender],
128           int256(finalAmount));
129
130         // Send Ether (with no gas)
131         msg.sender.call.value(finalAmount).gas(0);
132     }
133
134     // Transfers the given amount from the holder to the counter-
    party
135     function transferToHolder(int256 amount) private {
136         stakes[holder] = safeAddSigned(stakes[holder], amount);
137         stakes[counterParty] = safeSubSigned(stakes[counterParty],
138           amount);
139     }
140
141     // Add two signed integers if no overflow or underflow can occur
142     function safeAddSigned(int256 a, int256 b) private view
143       returns (int256) {
144         require(
145             (b >= 0 && a <= MAX_INT256 - b) ||
146             (b < 0 && a >= MIN_INT256 - b),
147             "Integer overflow or underflow."
148         );
```

```
149
150         return a + b;
151     }
152
153     // Subtract one signed integer from another if no overflow or
    underflow can occur
154     function safeSubSigned(int256 a, int256 b) private view
155       returns (int256) {
156         require(
157             b != MIN_INT256,
158             "Integer overflow or underflow."
159         );
160
161         return safeAddSigned(a, -b);
162     }
163 }
```

Listing B.2: Solidity Implementation of 'scaleK 100 (get (truncate "01/01/2020 12:00:00" (one))) [61]'

B.3 European Options

```
pragma solidity >=0.4.22 <0.6.0;

contract EuroOption {
    // A struct representing an or-choice
    struct OrChoice {
        // Whether or not the or-choice has been set
        bool set;

        // The or-choice (true is the first sub-contract,
        // false is the second sub-contract)
        bool choice;
    }

    // Static values
    int256 MAX_INT256 = int256(~(uint256(1) << 255));
    int256 MIN_INT256 = int256(uint256(1) << 255);
    uint256 HORIZON_UNIX = 1577880000;

    // The contract holder
    address holder;

    // The counter-party
    address counterParty;

    // The stakes of the holder and counter-party
    mapping(address => int256) stakes;

    // Whether or not this contract has been acquired
    bool acquired;

    // The or-choice
    OrChoice orChoice = OrChoice(false, false);

    // Constructor, takes the contract holder address
    constructor(address contractHolder) public {
        require(
            contractHolder != msg.sender,
            "Holder and counter-party cannot have the same address."
        );
        // Set the holder and counter-party
        holder = contractHolder;
        counterParty = msg.sender;

        // Initialise stakes to 0
        stakes[counterParty] = 0;
        stakes[holder] = 0;
        acquired = false;
```

```
48      }
49
50      // Only allows the holder or counter-party to call a function
51      modifier onlyParties() {
52          require(
53              msg.sender == counterParty || msg.sender == holder,
54              "can only be called by the holder or counter-party."
55          );
56
57          _;
58      }
59
60      // Only allows the holder to call a function
61      modifier onlyHolder() {
62          require(
63              msg.sender == holder,
64              "Only the holder may call this function."
65          );
66
67          _;
68      }
69
70      // Returns the balance of one of the two parties
71      function getBalance(bool holderBalance) public view
72        returns (int256) {
73          if (holderBalance) {
74              return stakes[holder];
75          } else {
76              return stakes[counterParty];
77          }
78      }
79
80      // Sets the choice for the or-combinator
81      function setOrChoice(bool firstSubContract) public
82        onlyHolder() {
83          require(
84              !orChoice.set,
85              "The or-choice has already been set."
86          );
87
88          orChoice.set = true;
89          orChoice.choice = firstSubContract;
90      }
91
92      // Acquires this contract
93      function acquire() public onlyHolder() {
94          require(
95              !acquired,
96              "This function can only be called before acquisition."
97          );
98          require(
```

```
99              now <= HORIZON_UNIX,
100             "This contract can only be acquired until
101               01/01/2020 12:00:00 UTC."
102         );
103
104         acquired = true;
105
106         require(
107             orChoice.set,
108             "The or-choice must be set before updating."
109         );
110
111         if (now < HORIZON_UNIX) {
112             return;
113         }
114
115         // If or-choice is true, acquire one
116         if (orChoice.choice) {
117             transferToHolder(1);
118         }
119     }
120
121     // Stake Ether in the contract
122     function stake() public payable onlyParties() {
123         require(
124             uint256(MAX_INT256) >= msg.value,
125             "Value staked is too big to be stored as int256."
126         );
127
128         // Update balance
129         stakes[msg.sender] = safeAddSigned(stakes[msg.sender],
130           int256(msg.value));
131     }
132
133     // Withdraw Ether from the contract
134     function withdraw(uint64 amount) public onlyParties() {
135         require(
136             address(this).balance > 0,
137             "Contract does not have enough funds."
138         );
139         require(
140             stakes[msg.sender] > 0,
141             "The caller does not have enough stake."
142         );
143
144         uint64 finalAmount = amount;
145
146         // Clamp withdrawal amount to total contract balance
147         if (address(this).balance < finalAmount) {
148             finalAmount = uint64(address(this).balance);
149         }
```

```
150
151         // Clamp withdrawal amount to party's balance
152         if (stakes[msg.sender] < finalAmount) {
153             finalAmount = uint64(stakes[msg.sender]);
154         }
155
156         // Adjust balance first to prevent re-entrancy bugs
157         stakes[msg.sender] = safeSubSigned(stakes[msg.sender],
158           int256(finalAmount));
159
160         // Send Ether (with no gas)
161         msg.sender.call.value(finalAmount).gas(0);
162     }
163
164     // Transfers the given amount from the holder to the counter-
    party
165     function transferToHolder(int256 amount) private {
166         stakes[holder] = safeAddSigned(stakes[holder], amount);
167         stakes[counterParty] = safeSubSigned(stakes[counterParty],
168           amount);
169     }
170
171     // Add two signed integers if no overflow or underflow can occur
172     function safeAddSigned(int256 a, int256 b) private view
173       returns (int256) {
174         require(
175             (b >= 0 && a <= MAX_INT256 - b) ||
176             (b < 0 && a >= MIN_INT256 - b),
177             "Integer overflow or underflow."
178         );
179
180         return a + b;
181     }
182
183     // Subtract one signed integer from another if no overflow or
    underflow can occur
184     function safeSubSigned(int256 a, int256 b) private view
185       returns (int256) {
186         require(
187             b != MIN_INT256,
188             "Integer overflow or underflow."
189         );
190
191         return safeAddSigned(a, -b);
192     }
193 }
```

Listing B.3: Solidity Implementation of 'get (truncate "01/01/2020 12:00:00" (one or zero)) [61]'

Appendix C

User Manual

This chapter will walk through the actions that need to be taken in order to run the system on a local machine. It is divided into three sections. Firstly, we will explain how to install the software and required libraries. Following this, we will explain how to test, build and run the software on a local server. Lastly, we will go on to elaborate on how to interact with the web application and compose a Nexus smart contract. The given instructions are short — for more details on how to use our software, the GitHub repository at **https://github.com/Noah-Vincenz/Financial-Smart-Contracts.git** offers a more detailed set of instructions. Additionally, it should be noted that the commands listed in this chapter are macOS terminal commands. For usage instructions on a different operating system please refer to the aforementioned GitHub repository. First of all, the instructions in the tutorial at **https://github.com/paritytech/pwasm-tutorial** should be followed to set up a small pWasm project, install the libraries and tools that are required for this project, and become familiar with the pWasm and Rust environment.

C.1 Installation & Setup

C.1.1 Software

In order to download the software locally one can head to the GitHub repository linked above and clone the repository onto the machine that is being used. You can then head to the root directory of the repository and run

```
$ npm install
```

This will install all missing npm libraries that are needed to run the software. This assumes that npm is installed on the machine being used. If this is not the case, the instructions on the npm website **https://www.npmjs.com/get-npm** can be followed to to install the missing tools.

C.1.2 Libra

In order to install Libra, one needs to head to **https://github.com/libra/libra** and clone the repository. This will download the Libra repository onto the machine that is being used. Once the download has completed, the user can change directory to the root directory of the repository. Following this, one can run Libra locally in the terminal by running

```
$ ulimit -n 4096
$ cargo run -p libra_swarm -- -s
```

This will launch a single Libra validator node locally on your own blockchain. The running node, however, is not connected to the Libra testnet. This allows you to play around with Libra accounts and exercise the functionality offered by the Move IR. Furthermore, this can be used to send transactions that publish a module, run the transaction script and so one. As of now, the documentation for this is sparse and the Libra Association are currently working on supplying more functionality.

In addition to the above, one can change directory to libra/language/functional_tests /tests/testsuite to verify the semantics and correctness of any given Libra code, allowing you to exercise modules that modify the global blockchain state in the same way you could do on a real blockchain. After the Move IR source code file named *script.mvir*, which was produced in section 6.2.6, has been successfully downloaded, this can be located inside the directory above. Running

```
$ cargo test script.mvir
```

will then execute the transactions in the downloaded module and verify its correctness.

C.2 Project Test, Build, and Run

After the software has been downloaded and required libraries have been installed, you can test the software by running

```
$ npm run test-js
```

to execute the given set of JavaScript tests, or

```
$ npm run test-rust
```

to execute our Rust tests. Running

```
$ npm run build
```

will compile the Rust smart contract into its corresponding JSON contract code using Wasm. Following this, running

```
$ gulp
```

will bundle the JavaScript files and launch a local server hosted at **localhost:9001** to execute these and launch our web application. Before accessing the web application, the user must run

```
$ ./run-parity-chain.sh
```

in a separate terminal window in order to run a local blockchain instance of the parity development chain. Heading to **localhost:9001** in Google Chrome will give you access to the we application. Alternatively, you can run

```
$ npm start
```

to execute all the steps mentioned above in their order given.

C.3 Using the Web Application

The web application can be accessed at **localhost:9001** using Google Chrome. The instructions in this section assume that MetaMask is correctly installed and set up. If this is not the case, you can head to **https://metamask.io/** to follow their instructions to install MetaMask, import the parity development blockchain with **http://127.0.0.1:8545** as the RPC URL, and import the accounts registered on the local parity development blockchain. Alternatively, refer to the aforementioned GitHub repository for instructions. When running the application for the first time or on restarting the browser, the user will be presented with a MetaMask window, asking to allow the web application to access MetaMask. After having confirmed this, the selected network in MetaMask should be switched to the parity development blockchain network and one of the imported blockchain accounts should be selected. Once these instructions have been followed and the web application is up and running, the user will be presented with the following user interface.

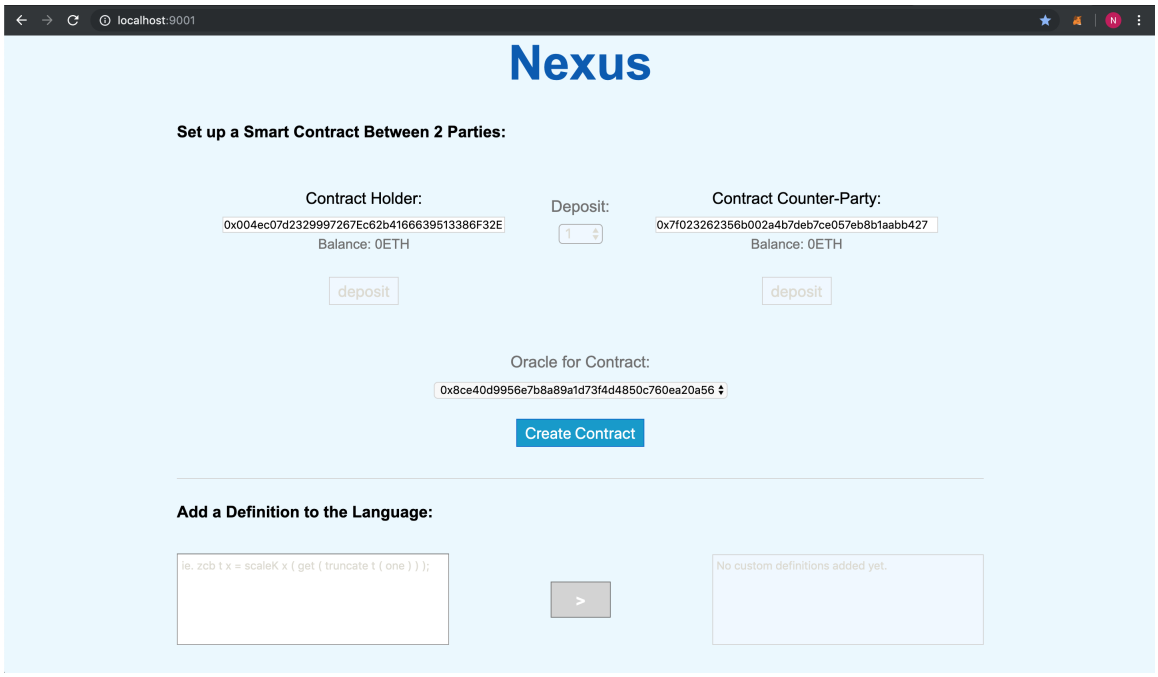


Figure C.1: Web UI Screenshot 1

The user can then provide two parity development chain account addresses to be used for the contract and press the *Create Contract* button to proceed. This will trigger MetaMask showing the window presented below, asking the user to confirm the transaction.

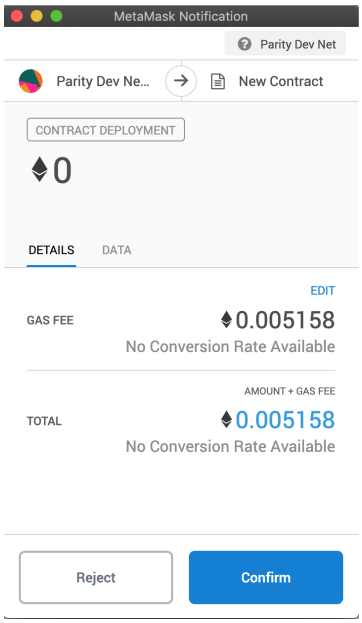


Figure C.2: MetaMask Transaction Screenshot

Confirming this transaction will enable the *deposit* buttons in the web user interface, allowing the user to deposit a specified amount of Ether into both accounts. In order to deposit Ether, the account that is placing the deposit must be selected in MetaMask, otherwise the user will be notified with an error message in the user interface. After both accounts have deposited an arbitrary amount of Ether, the input textarea titled *Construct Smart Contract Transactions:* will be enabled and the user can start composing Nexus contracts by providing a syntactically correct contract from the textarea and then pressing the *Make Transaction* button. This will add the contract provided by the user to the list of pending contracts and display this in the table presented in figure C.3. In the following figure, the most recently added contract corresponds to **zero and give (zero)**, where both subcontracts are displayed as a single combined supercontract.

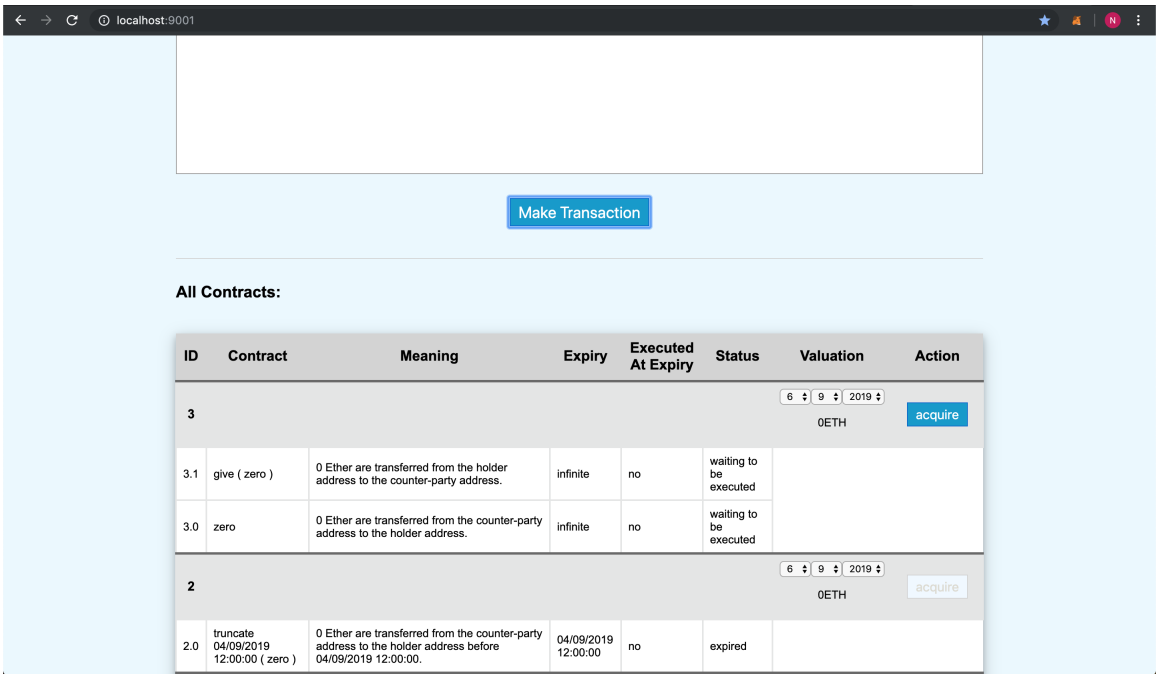


Figure C.3: Web UI Screenshot 2

Following this, the user can freely use the web application interface to extend the language, add new contracts, evaluate and manage contracts or acquire pending contracts.

Bibliography

[1] S. P. Jones, J.-M. Eber, and J. Seward, "Composing contracts: An adventure in financial engineering," *Lecture notes in computer science.*, vol. 2021, 2001.

[2] Y. Khatri, "World Bank, CommBank Team Up for 'World First' Blockchain Bond Transaction," https://www.coindesk.com/world-bank-commbank-team-up-for-world-first-blockchain-bond-transaction, 2019, [Online; accessed 28-May-2019].

[3] H. Partz, "BBVA Signs $117 Mln Blockchain-Powered Corporate Loan," "https://cointelegraph.com/news/bbva-signs-117-mln-blockchain-powered-corporate-loan", 2018, [Online; accessed 28-May-2019].

[4] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," "http://bitcoin.org/bitcoin.pdf", 2008, [Online; accessed 28-May-2019].

[5] S. Haber and W. S. Stornetta, "How to Time-stamp a Digital Document," *Journal of Cryptology*, vol. 3, pp. 99–111, 1991.

[6] N. Szabo, "Smart Contracts," http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html, 1994, [Online; accessed 28-May-2019].

[7] M. J. S. Smith, "Application-Specific Integrated Circuits," http://d1.amobbs.com/bbs_upload782111/files_9/ourdev_212152.pdf, [Online; accessed 24-August-2019].

[8] D. Middleton, "Meet Sawtooth Lake," https://www.hyperledger.org/blog/2016/11/02/meet-sawtooth-lake, [Online; accessed 24-August-2019].

[9] P. Foundation, "Privacy-focused & Decentralized Ecosystem," https://particl.io/, [Online; accessed 23-August-2019].

[10] B. Studios, "Yes, Bitcoin Can Do Smart Contracts and Particl Demonstrates How," https://bitcoinmagazine.com/articles/yes-bitcoin-can-do-smart-contracts-and-particl-demonstrates-how, [Online; accessed 23-August-2019].

[11] M. Chand, "What is the Most Popular Blockchain in the World?" https://www.c-sharpcorner.com/article/what-is-the-most-popular-blockchain-in-the-world2/, [Online; accessed 23-August-2019].

[12] V. Buterin, "Ethereum White Paper: A next-generation smart contract and decentralized application platform," https://github.com/ethereum/wiki/wiki/White-Paper, [Online; accessed 23-August-2019].

[13] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," http://gavwood.com/Paper.pdf, [Online; accessed 23-August-2019].

[14] Y. Sompolinsky and A. Zohar, "Accelerating bitcoin's transaction processing. fast money grows on trees, not chains," *IACR Cryptology ePrint Archive*, vol. 2013, p. 881, 2013.

[15] V. Buterin, "Dagger: A memory-hard to compute, memory-easy to verify scrypt alternative," http://www.hashcash.org/papers/dagger.html, [Online; accessed 24-August-2019].

[16] T. Dryja, "Hashimoto: I/o bound proof of work," http://diyhpl.us/~bryan/papers2/bitcoin/meh/hashimoto.pdf, [Online; accessed 24-August-2019].

[17] V. Buterin and V. Griffith, "Casper the friendly finality gadget," *ArXiv*, vol. abs/1710.09437, 2017.

[18] Ethereum, "Visibility and Getters," https://solidity.readthedocs.io/en/v0.4.24/contracts.html#visibility-and-getters, [Online; accessed 24-August-2019].

[19] ——, "Functions," https://solidity.readthedocs.io/en/v0.4.24/contracts.html#functions, [Online; accessed 24-August-2019].

[20] ——, "Contracts," https://solidity.readthedocs.io/en/v0.4.24/contracts.html, [Online; accessed 24-August-2019].

[21] I. Sergey and A. Hobor, "A concurrent perspective on smart contracts," *CoRR*, vol. abs/1702.05511, 2017. [Online]. Available: http://arxiv.org/abs/1702.05511

[22] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts," https://eprint.iacr.org/2016/1007.pdf, [Online; accessed 25-August-2019].

[23] M. Trillo, "Stress Test Prepares VisaNet for the Most Wonderful Time of the Year," https://www.visa.com/blogarchives/us/2013/10/10/stress-test-prepares-visanet-for-the-most-wonderful-time-of-the-year/index.html, 2013, [Online; accessed 24-July-2019].

[24] Visa, "Security and reliability," https://usa.visa.com/run-your-business/small-business-tools/retail.html, 2013, [Online; accessed 24-July-2019].

[25] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena, "A secure sharding protocol for open blockchains," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. the 2016 ACM SIGSAC Conference. New York NY: ACM, 2016, pp. 17–30.

[26] J. Poon and T. Dryja, "The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments," https://lightning.network/lightning-network-paper.pdf, 2016, [Online; accessed 24-July-2019].

[27] L. A. Members, "An Introduction to Libra," https://libra.org/en-US/white-paper/, 2019, [Online; accessed 22-July-2019].

[28] L. Association, "Welcome to Libra," https://libra.org/en-US/, [Online; accessed 25-July-2019].

[29] M. Baudet, A. Ching, A. Chursin, G. Danezis, F. Garillot, Z. Li, D. Malkhi, O. Naor, D. Perelman, and A. Sonnino, "State Machine Replication in the Libra Blockchain," https://developers.libra.org/docs/assets/papers/libra-consensus-state-machine-replication-in-the-libra-blockchain.pdf, 2019, [Online; accessed 22-July-2019].

[30] M. Santori, https://twitter.com/msantoriESQ/status/1151526693670150145, 2019, [Online; accessed 23-July-2019].

[31] I. A. Kakadiaris, A. Kumar, W. J. Scheirer, and J. V. Monaco, "Identifying bitcoin users by transaction behavior," in *Proceedings of SPIE–the international society for optical engineering.*, ser. SPIE Defense + Security, vol. 9457. Bellingham, Wash. :: Society of Photo-optical Instrumentation Engineers, 2015.

[32] L. Association, "Libra Core Overview," https://developers.libra.org/docs/libra-core-overview, [Online; accessed 25-July-2019].

[33] Z. Amsden, R. Arora, S. Bano, M. Baudet, S. Blackshear, A. Bothra, G. Cabrera, C. Catalini, K. Chalkias, E. Cheng, A. Ching, A. Chursin, G. Danezis, G. D. Giacomo, D. L. Dill, H. Ding, N. Doudchenko, V. Gao, Z. Gao, F. Garillot, M. Gorven, P. Hayes, J. M. Hou, Y. Hu, K. Hurley, K. Lewi, C. Li, Z. Li, D. Malkhi, S. Margulis, B. Maurer, P. Mohassel, L. de Naurois, V. Nikolaenko, T. Nowacki, O. Orlov, D. Perelman, A. Pott, B. Proctor, S. Qadeer, Rain, D. Russi, B. Schwab, S. Sezer, A. Sonnino, H. Venter, L. Wei, N. Wernerfelt, B. Williams, Q. Wu, X. Yan, T. Zakian, and R. Zhou, "The Libra Blockchain," https://developers.libra.org/docs/assets/papers/the-libra-blockchain.pdf, 2019, [Online; accessed 22-July-2019].

[34] L. Association, "Libra Protocol: Key Concepts," https://developers.libra.org/docs/libra-protocol, [Online; accessed 25-July-2019].

[35] I. Abraham, G. Gueta, and D. Malkhi, "Hot-Stuff the Linear, Optimal-Resilience, One-Message BFT Devil," https://dahliamalkhi.files.wordpress.com/2018/03/hot-stuff-arxiv2018.pdf, 2018, [Online; accessed 25-July-2019].

[36] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM transactions on programming languages and systems*, vol. 4, no. 3, pp. 382–401, 1982.

[37] S. Micali, M. O. Rabin, and S. P. Vadhan, "Verifiable random functions," *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*, pp. 120–130, 1999.

[38] S. Blackshear, E. Cheng, D. L. Dill, V. Gao, B. Maurer, T. Nowacki, A. Pott, S. Qadeer, Rain, D. Russi, S. Sezer, T. Zakian, and R. Zhou, "Move: A Language With Programmable Resources," https://developers.libra.org/docs/assets/papers/libra-move-a-language-with-programmable-resources.pdf, 2019, [Online; accessed 22-July-2019].

[39] L. Association, "Libra's mission is to enable a simple global currency and financial infrastructure that empowers billions of people," https://github.com/libra/libra, [Online; accessed 27-July-2019].

[40] L. Reyzin, S. Yakoubov, V. Zikas, and R. D. Prisco, *Efficient Asynchronous Accumulators for Distributed PKI*, ser. Security and Cryptography for Networks. Berlin ;: Springer-Verlag, 2016, vol. 9841.

[41] P. O. Stake, "Proof of Stake and Scalability," https://medium.com/@poolofstake/proof-of-stake-and-scalability-2ecaa8fd5d7c, [Online; accessed 25-July-2019].

[42] S. Bano, C. Catalini, G. Danezis, N. Doudchenko, B. Maurer, A. Sonnino, and N. Wernerfelt, "Moving Toward Permissionless Consensus," https://libra.org/en-US/permissionless-blockchain/#overview, 2019, [Online; accessed 24-July-2019].

[43] T. Guardian, "Revealed: 50 million Facebook profiles harvested for Cambridge Analytica in major data breach," https://www.theguardian.com/news/2018/mar/17/cambridge-analytica-facebook-influence-us-election, 2019, [Online; accessed 22-July-2019].

[44] J. Biggs, "Watch the Facebook Libra Hearing Live," https://www.coindesk.com/watch-the-facebook-libra-hearing-live-now, 2019, [Online; accessed 23-July-2019].

[45] ConsenSys, "Smart Contract Security Best Practices," https://github.com/ConsenSys/smart-contract-best-practices/blob/master/docs/recommendations.md, 2016, [Online; accessed 29-May-2019].

[46] S. Kapoor, "What is SolidityX?" https://medium.com/@shivaanshkapoor02/what-is-solidityx-92624e30b48, 2019, [Online; accessed 29-May-2019].

[47] S. Akinyemi, "Awesome webassembly languages," https://github.com/appcypher/awesome-wasm-langs, [Online; accessed 25-August-2019].

[48] A. V. Deursen and P. Klint, "Little languages: little maintenance?" *Journal of Software Maintenance: Research and Practice*, vol. 10, no. 2, pp. 75–92, 1998. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=61CD154A08E7CCEFA1E6D9F80685FB8F?doi=10.1.1.50.4726&rep=rep1&type=pdf

[49] F. Schrans, D. Hails, A. Harkness, S. Drossopoulou, and S. Eisenbach, "Flint for safer smart contracts," https://arxiv.org/pdf/1904.06534.pdf, [Online; accessed 25-August-2019].

[50] BitcoinWiki, "Lisk," https://en.bitcoinwiki.org/wiki/Lisk, 2016, [Online; accessed 29-May-2019].

[51] D. Asset, "The digital asset platform," https://hub.digitalasset.com/hubfs/Documents/Digital%20Asset%20Platform%20-%20Non-technical%20White%20Paper.pdf, [Online; accessed 25-August-2019].

[52] ——, "Digital asset and blockchain technology partners to deliver daml smart contracts for hyperledger sawtooth," https://hub.digitalasset.com/hubfs/Press%20Releases/DAML%20on%20Hyperledger%20Sawtooth.pdf, [Online; accessed 25-August-2019].

[53] Hyperledger, "Hyperledger sawtooth," https://www.hyperledger.org/projects/sawtooth, [Online; accessed 25-August-2019].

[54] WebAssembly, "WebAssembly," https://webassembly.org, 2016, [Online; accessed 30-May-2019].

[55] Ethereum, "Parity," https://github.com/ethereum/homestead-guide/blob/master/source/ethereum-clients/parity/index.rst, 2017, [Online; accessed 31-May-2019].

[56] ——, "Sharding Roadmap," https://github.com/ethereum/wiki/wiki/Sharding-roadmap, 2018, [Online; accessed 31-May-2019].

[57] A. Neumann, "Writing a Simple Parser in Rust," https://adriann.github.io/rust_parser.html, 2017, [Online; accessed 19-July-2019].

[58] S. Y. Cheung, "The dangling else ambiguity," http://www.mathcs.emory.edu/~cheung/Courses/561/Syllabus/2-C/dangling-else.html, [Online; accessed 13-August-2019].

[59] A. V. Aho and S. C. Johnson, "Lr parsing," *ACM Computing Surveys*, vol. 6, no. 2, pp. 99–124, Jun. 1974. [Online]. Available: http://doi.acm.org/10.1145/356628.356629

[60] N.-V. Noeh, "Running main function of a .mvir file from terminal," https://github.com/libra/libra/issues/115, [Online; accessed 4-September-2019].

[61] D. Dean, "Smartfin - implementing a financial domain-specific language for smart contracts," https://github.com/danrobdean/SmartFin/tree/master/report-solidity-contracts, [Online; accessed 4-September-2019].

[62] P. Koronkevich, "Obsidian in the rough: A case study evaluation of a new blockchain programming language," https://obsidian-lang.com/obsidian-case-study.pdf, [Online; accessed 6-September-2019].

[63] C. I. Client, "Ho-lee model," http://help.cqg.com/cqgic/default.htm#!Documents/holeemodel.htm, [Online; accessed 6-September-2019].