HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

COMPUTER SCIENCE

Improving the Bazo Blockchain

Term Project

Department of Computer Science
University of Applied Science Rapperswil

Autumn Term 2018

Author(s): Remo Pfister, Keerthikan Thurairatnam
Advisor: Prof. Dr. Thomas Bocek

Abstract

The study work focuses on designing a smart contract language (named "Lazo") for the Bazo blockchain. The Bazo blockchain is a research blockchain to test different mechanisms and algorithms. In the current version, a *Proof of Stake* consensus algorithm and a virtual machine to execute Bazo intermediate language (opcodes) are integrated. However, writing smart contracts in Bazo opcodes is time consuming and error-prone. The goal of this study work is to design a high-level language which is easier to read and write smart contracts.

Before designing Lazo, 24 existing smart contract languages are collected and roughly analyzed to identify the key characteristics of a language for the blockchain. Thereafter, three popular and well elaborated languages, namely Solidity, Vyper and Scilla, were analyzed in great detail. Their supported features, syntax and contract examples were also documented. With the acquired knowledge about smart contracts, Lazo language was designed in an agile manner.

As a result, Lazo is designed to be a statically typed, imperative and non-turing complete programming language. All language features are documented with illustrative code snippets. The Lazo grammar is also written in ANTLR and verified with Java. Furthermore, contract examples from Solidity are translated to Lazo in order to prove that the real-world use cases can be programmed with Lazo as well.

In a follow-up thesis, a compiler could be developed to compile Lazo programs into Bazo virtual machine instructions.

Management Summary

Initial Situation The Bazo blockchain is a research blockchain to test different mechanisms and algorithms. In the current version, a *Proof-of-Stake* consensus algorithm and mechanisms to run the blockchain on mobile devices are integrated. Furthermore, there is also a virtual machine available to interpret and execute intermediate language (opcodes) on the Bazo blockchain. However, writing smart contracts in Bazo opcodes is time consuming and error-prone. The goal of this study work is to specify a smart contract language for the Bazo blockchain, so that it is easier to read and write smart contracts.

Procedure The language design project consisted of four sub goals, namely rough analysis, detailed analysis, language specification and verification. In the rough analysis, 24 existing smart contract languages were roughly analyzed and typical characteristics of a smart language were identified. In the detailed analysis, Solidity, Vyper and Scilla were thoroughly analyzed and their features and syntax were also documented. Furthermore, famous or frequent attacks on Solidity were analyzed and countermeasures were taken into consideration in the Lazo language. During the language specification phase, all language features were specified in close consultation with our supervisor. Finally, the language syntax was verified with 117 test cases.

Result The Lazo language is designed with the aim of achieving better readability and high robustness. As a result, Lazo is a statically typed, imperative and non-turing complete programming language. Even though Lazo is inspired by Solidity, many unnecessary features are removed and essential features are simplified when needed, thus making the contracts easier to understand. Security concerns are also taken into account and countermeasures are built-in at language level, where possible.

Outlook According to the language specification, a compiler could be developed to compile Lazo programs into Bazo virtual machine instructions (opcodes). If there are no opcodes available for certain new features, the Bazo VM needs to be extended.

Acknowledgments

At this point, we would like to thank everyone involved in the Lazo project.

Obviously, our special thanks goes to our supervisor Prof. Dr. Thomas Bocek, who was very supportive throughout the whole project. He had a great deal of experience with blockchain and contract programming and gave us many valuable inputs to improve the language. Designing a new smart contract programming language in close consultation with him also gave us a great insight into how blockchain and smart contracts work.

We would also like to thank other students from University Zurich and HSR for their work on the Bazo blockchain. We are very grateful for having this opportunity and being able to make some contribution to the open source Bazo research blockchain technology.

Contents

Part I

Context

Chapter 1

Introduction

The Bazo Blockchain is a Blockchain Research Project started at the University of Zurich in 2017. Currently, writing smart contracts for the Bazo Blockchain is possible but rather hard as no high-level programming language exists. In this document we are going to cover all the stages required to design a high-level contract language for the Bazo Blockchain.

1.1 Motivation

In the bachelor thesis 'Integrating Smart Contracts into the Bazo Blockchain', Ennio Meier and Marco Steiner created a virtual machine for the bazo blockchain to be able to interpret smart contracts for the bazo blockchain. To test it, they created a low-level smart contract language based on Operation codes (Opcodes) which is rather very hard to understand. To make writing smart contracts for the bazo blockchain user friendly, a new language is required. It should be easy to read and write but at the same time should be also powerful enough to handle all the use cases. With a good smart contract language based on well known and established paradigms and syntax, users will be able to write contracts without the need to invest a lot of time to learn the language in advance.

1.2 Description of Work

The goal of this work is to design a contract language for the Bazo Blockchain based on already existing solutions. We are focusing on Solidity, a very well established smart contract

language for the Ethereum Blockchain. To gain as much knowledge about contract languages as possible, it is necessary to analyze other existing languages in detail. To be able to do so, we first need to get an overview of which languages exist and what their characteristics are. After that we will choose three of those languages, analyze them in detail and from there we are going to design our own language. The new language should be lightweight and easy to read and write. As a result a grammar will be created and verified using ANother Tool for Language Recognition (ANTLR) and Java.

Chapter 2

Background and Related Work

In the section 'Background', the technologies used for the Bazo Blockchain are described. The section 'Related Work' lists theses of other students who worked on it as well as similar independent projects.

2.1 Background

2.1.1 Blockchain

A Blockchain basically consists of blocks of transactions which are chained one after the other. It is a distributed, transactional database. Everyone in the network can read entries from this database. Entries in the database are immutable. The database can only be extended by creating new transactions. To make the transactions non-repudiable, digital signatures are used. Transactions are validated by miners. To verify a transaction, the miner validates the signature and checks if the assets being transmitted actually belong to the sender.

2.1.2 Smart Contracts

Smart contracts can be understood as real world contracts or agreements written in computer code. Those contracts are stored on the blockchain. A user can interact with the contract by creating new transactions and sending them to the contract address. Those contracts are executed by a virtual machine.

2.1.3 Transactions

The following definition can be applied to blockchain: "In the context of data base management systems a transaction is a unit of work performed within the system."[1] There are multiple types of transactions but we will not go into further detail in this work. Please refer to the bachelor thesis "Integrating Smart Contracts into the Bazo Blockchain" for further information.

2.1.4 Virtual Machine

A virtual machine is basically an abstraction of hardware. Instead of writing code directly for the underlying hardware, users can write code for the virtual machine, which is a lot easier and less error prone.

2.1.5 The Bazo Blockchain

In 2017 the Bazo Blockchain was started as a research project at the University of Zurich. Currently, it consists of the following components:

Miner Is used to run a full bazo blockchain node. The Miner validates new transactions and records them in the global ledger.

Client A command line interface to interact with the Bazo blockchain.

Wallet A web-based wallet for the bazo cryptocurrency.

Block Explorer The block explorer is a web-based graphical user interface to gain insights into the blockchain by making the blocks and the transactions visibe.

Virtual Machine The virtual machine interpretes smart contracts written for the bazo blockchain and executes them on the miner.

Note: The information above have been taken from the official Bazo Blockchain Github Repo [2]

For further detail about the Bazo Blockchain consult either the official Bazo Blockchain Github Repository [2] or the 'Integrating Smart Contracts into the Bazo Blockchain' documentation [1].

2.2 Related Work

2.2.1 Previous Work

As already mentionned, the bazo blockchain was started in 2017. Since then, several people have further developed the project.

Bazo - A Cryptocurrency from Scratch August 2017 (UZH), Livio Sgier

A Progressive Web App (PWA)-based Mobile Wallet for Bazo January 2018 (UZH), Jan von der Assen

A Blockchain Explorer for Bazo January 2018 (UZH), Luc Boillat

Proof of Stake for Bazo January 2018 (UZH), Simon Bachmann

Design and Prototypical Implementation of a Mobile Light Client for the Bazo Blockchain January/March 2018 (UZH), Marc-Alain Chételat

Cryptographic Sortition for Proof of Stake in Bazo May 2018, Roman Blum

A pruneable approach for Bazo August 2018 (HSR), Stefano Fontana

Integrating Smart Contracts into the Bazo Blockchain Spring Term 2018 Ennio Meier, Marco Steiner

2.2.2 Existing Solutions

NEO "NEO is a blockchain project «that utilizes blockchain technology and digital identity to digitize assets, to automate the management of digital assets using smart contracts, and to realize a smart economy with a distributed network.» NEO utilizes a consensus mechanism called the Delegated Byzantine Fault Tolerance. NEO is implemented in C#." [1]

Ethereum "The goal of Ethereum is to create a platform for the development of decentralized apps in order to create a «more globally accessible, more free, and more trustworthy Internet, an internet 3.0». There are several implementations of the client such as go-ethereum (written in Go), cpp-ethereum (written in C++) and others. Ethereum's

consensus mechanism is Proof of Work but a Proof of Stake algorithm is already being developed and likely to go live in 2018." [1]

Bitcoin Bitcoin is a digital currency based on blockchain technologies. It was the first real implementation of the blockchain technology and gained a lot of attention. Proof of Work is used as the consensus mechanism and mining a block takes about 10 minutes.

Part II

Language Design

Chapter 3

Language Characteristics

Lazo is a compiled and contract-oriented programming language for the Bazo Blockchain. The goals of the language are to be simple, expressive and secure in writing reliable and solid smart contracts. Lazo is similar to Solidity. It borrows and adapts good concepts from Solidity while avoiding features that have led to complexity and unreliable code.

Lazo source code will be compiled to Bazo Intermediate Language (IL). Bazo IL consists of Opcodes that run on the stack-based Bazo Virtual Machine (VM).

3.1 Programming Paradigms

Lazo is a multi-paradigm programming language.

- **Imperative programming**: Instructions are explicit. Computation takes place step by step and change the program state.
- **Contract-oriented programming**: Data fields (states) and methods are encapsulated into contract objects. These objects can communicate with each other using the public interfaces.

3.2 Type System

Lazo is a **statically** typed language, i.e. verifies the type safety of the program during the compilation time. Therefore, contracts are less prone to type errors than in dynamic languages at

run-time.

Lazo uses the **nominal typing system**. Two variables are type-compatible, if and only they have the same type, like in Java.

Lazo has **no type inferences**, meaning there is no automatic type detection at compile time. Programmers should declare the types explicitly. Furthermore, Lazo does not support **implicit type conversions** either. In order to concatenate a string and a number, the number should be explicitly converted to a string first.

Category	Lazo
Static vs Dynamic	Static
Nominal vs Structural	Nominal
Manifest vs Inference	Manifest

Table 3.1: Overview of Type System

3.3 TURING COMPLETENESS

Bazo VM is **turing-complete** and allows to create endless loop [1]. For Lazo to become turing-complete as well, it should support endless loop in at least one of the following forms:

- Indefinite control flow statement (e.g. *while*-loop over a variable)
- Recursion
- Cyclic contract calls

Lazo, however, is **not turing-complete** because it does not support any of the above features at the language level (See sections *6.9, 6.10 and 7*). The reason for that is we want to prove that programs will stop at a certain point. Also gas limit attacks can be avoided, but it does not protect from poorly written code or not providing enough gas at all.

Nevertheless, Lazo does allow a simple limited *for*-loop, the range of which is pre-determined.

3.4 CHARACTER SET AND ENCODING

Lazo source files are encoded in American Standard Code for Information Interchange (ASCII) format, therefore only valid English characters are allowed in character or string literals.

3.5 MAJOR OMISSIONS

Lazo deliberately omits the following language features in the first version.

- Inheritance
- Abstract contracts
- Generics
- Libraries
- Currency Units
- ABI Encoding Functions
- Function Modifiers
- Function Overloading
- Recursion
- Inline Assembly
- Self-destruct

Chapter 4

Program

A Lazo program must contain exactly **one** contract in a single file. However, it can contain one or more interfaces. Furthermore, the program does not support importing any other files or libraries. Therefore, everything should be programmed within the same file.

4.1 A Simple Program

```
1  version 1.0
2
3  contract SimpleContract{
4      Map<address, int> payments
5
6      [Payable]
7      [Pre: msg.coins > 0]
8      function void pay() {
9          payments[msg.sender] += msg.coins
10     }
11 }
```

4.1.1 Version

Lazo supports versioning in the following form:

```
version 1.0 // <major>.<minor>
```

In Lazo, all minor versions are backward compatible with the defined major version. Symbols like caret ^ or tilde ~, as known from solidity, are not supported.

Versioning allows us to extend or modify the language (e.g. add/remove features) while still supporting contracts with old syntax. The developer decides for which version of the compiler their contracts are written for.

Examples:

- Version 1.1 is compatible with the compiler version 1.7
- Version 1.7 is not guaranteed to be compatible with the compiler version 1.1
- Version 1.7 is not guaranteed to be compatible with the compiler version 2.1

4.2 IDENTIFIERS

Identifiers start with a letter (a-zA-Z) or an underscore (_) followed by any arbitrary number of alphanumeric letters (a-zA-Z0-9) or underscores (_).

We recommend to write variable and function names in camel case (e.g. myName, getName) and all other identifiers in pascal case (e.g. MyContract) for readability. However, this is not enforced by the compiler.

```
int one = 1
int numberOfUsers = 5
String firstName = "Peter"

function void getFirstName() {
    // Do something
}
```

4.3 RESERVED KEYWORDS

Reserved keywords cannot be used as identifiers.

version	**contract**	**is**	**internal**	**function**
if/else	**foreach**	**for**	**emit**	**readonly**
continue	**break**	**event**	**return**	**constructor**
to	**by**	**throw**	**interface**	

Table 4.1: Program Keywords

int	**char**	**bool**
enum	**String**	**Map**
struct	**void**	**error**

Table 4.2: Type Keywords

true	**false**

Table 4.3: Constant keywords

Built-in function names are not allowed for keywords as well. Please refer to the corresponding section (6.7) for more information about built-in functions.

The following keywords are prohibited as they can be misleading or might be supported by the language in the future:

while	**goto**	**abstract**	**implements**	**external**
private	**ref**	**out**	**static**	**extends**
override	**virtual**	**as**	**const**	**var**
null	**public**	**switch**	**case**	**try**
catch	**finally**	**do**		

Table 4.4: Prohibited keywords

null is removed from the language, as otherwise developers need to perform null checks. Other languages (e.g. Solidity) do not support null either to simplify the language.

4.4 Declaration

4.4.1 Contract

A contract consists of fields, functions, events, struct and enum declarations. Contracts cannot be nested (no inner contracts). In a program file, there is only one contract declaration allowed.

4.4.2 Variable

Variables are declared as follows:

```
// Declaration with initialization
int one = 1
// Declaration without initialization
int numberOfUsers // default initialized with 0
```

The visibility of variable is always **internal** which means that the variables cannot be accessed from outside of the contract. This protects them from direct manipulation and helps the developer to avoid unintended behaviour by forcing him to create functions that modify the state.

If a declared variable is not used in the program, the compiler will throw an error. The reason for that is unused variables cost gas unnecessarily. This check might help the developers to save on some costs.

4.4.3 Constant

The keyword readonly can be used to declare constant variables. Those variables can only be initialized during the declaration or contract construction. They cannot be changed afterwards.

```
readonly int x = 0
readonly Person p = new Person("Peter")
```

For value types, readonly means that the value cannot change. For reference type, readonly means that the reference to the data location cannot change, but the underlying data can (e.g. name of a Person struct).

4.4.4 Scope

Lazo supports **block scoping**. This means that variables and functions are only accessible inside the enclosing block.

```
contract ScopeContract{
    int x = 3 // defined in the contract scope

    function int getResult(int a) {
        int x = 1 // creates a new variable x in the function scope
        return x  // returns the x defined within the function scope
    }

    function int getResultWithX() {
        int x = 1       // creates a new variable x in the function scope
        int y = this.x  // accesses the x in the contract scope
        return x + y
    }

    function int getX() {
        return x  // returns the x defined in the contract scope
    }
}
```

Hoisting is not supported by Lazo. This means that the following is not allowed:

```
function int getResult() {
    int y = x + 2   // x is used here but not yet declared
    int x = 5       // x is declared here
    return y
}
```

4.5 Statement Separation

Statements are separated by a newline "\n".

```
function int getResult(int a) {
    int b = 6
    int res = a + b
    return res
}
```

4.6 Indentation

Lazo does not validate indentation. However it is recommended to use 4 spaces as indentation.

4.7 Comments

Lazo supports both single-line and multi-line comments. Comments are used to enrich the source code with explanations such as why a certain design decision has been made. They should not be used too often, as they can make the code less readable.

4.7.1 Single-line Comments

Single-line comments are used to create comments that do not span more than one line. Everything before the comment is not interpreted as a comment. Everything after the comment-symbol '//' on the same line is interpreted as a part of the comment.

4.7.2 Multi-line Comments

Multi-line or block comments are used when a whole block should be interpreted as comment. Block comments are enclosed between '/*' (start of block comment) and '*/' (end of block comment).

4.7.2.1 Example

```
// This is a single-line comment!

/* This is a multi-line comment!
   Everything between the start and the end symbol is interpreted as comment!
*/
```

4.8 Global Variables

Global variables are stored as states in blockchain and are available throughout the whole program. They are read only and set by the program beforehand for the usage. Any assignments to the global variables will throw a compiler error.

We removed some global variables from Solidity when we could not find any useful use cases. The removed variables are also documented in the corresponding section.

4.8.1 msg

The transaction message is stored in the variable msg. It is important to note that the values of the msg fields can change, when an **external** function call is executed. The supported member fields are shown below:

- address **sender**: returns the address of the message sender. It is important that this value changes from call to call. To find out the initial sender, use tx.origin (see 4.8.3).
- int **coins**: returns the number of Bazo coins sent with the message. It is equivalent to *msg.value* in Solidity.
- int **sig**: returns the first four bytes of the hash from function signature, which is composed of visibility, return types, function identifier and arguments.
- int **gas**: returns the total gas allocated for the call.

Lazo does not support the **data** field from Solidity, which stores the complete raw call data. If any specific value from **data** is needed, it should be defined as fields with a suitable name for better usability.

4.8.2 block

The block variable stores the following information about the current block.

- int **number**: returns current block number
- int **timestamp**: returns the unix timestamp of the current block in seconds

The following field from Solidity are not supported.

- address **coinbase**: the address of the current block miner
- int **difficulty**: difficulty level
- int **gaslimit**: returns the total gas limit for the current block

4.8.3 tx

The initial transaction information is stored in tx. Even if several external function calls are performed, the information remains unchanged.

- int **gas**: return the total gas allocated for the transaction
- int **gasprice**: returns the total gas price of the transaction
- address **origin**: returns the initial sender of the transaction.

4.9 UNITS

Units are merely a shorthand version of writing another bigger integer number. They are quite convenient and give a meaning to a number. Units can be specified by adding suffixes to the literal numbers. Currently, Lazo supports only time units. Currency units are omitted for the first version.

```
1 int twoHours = 2_h              // equals to 7200 (seconds)
2 int duration = 5 * 3 * 1_min    // equals to 15 minutes or 900
```

Since it is an alias to an integer value, all integer operators(see 5.1) are allowed to be used.

4.9.1 Time Units

Unit Name	Unit Suffix	Equivalent Unit	Equivalent Integer Value
Second(s)	1_s	-	1
Minute(s)	1_min	60_s	60
Hour(s)	1_h	60_min	3'600
Day(s)	1_d	24_h	86'400
Week(s)	1_w	7_d	604'800

Table 4.5: Time Units

Note: The unit suffix 1_m is not used for minute, because m symbol is assigned to metre in Internation System of Units (SI). Apart from that, year unit is not supported because it is not always equal to 365 days because of the leap year.

Chapter 5

Types

Lazo is a statically typed language, i.e. the type of a variable should be declared explicitly. The types are divided into two categories: value types and reference types. A value type holds the actual data. When passing a value type, the data is copied. The reference type, on the other hand, stores the location of the real data. When passing a reference type, only the location of the data is copied, but not the data itself.

5.1 Value Types

5.1.1 Integer

Lazo only supports one integer size, namely big integer. The keyword int represents big integer and holds positive and negative integers. Default value is 0. Overflow checks are not required as the VM uses big integers for all the arithmetic operations.

```
int x = 5       // decimal value
int y = 0x5     // hexadecimal value
```

Integer Literal

In Lazo, integer literal is limited to 256 bits (32 bytes), However, it is possible to do calculations beyond this limit because they are stored in big integer type.

Operators

Operation	Operator Signs	Result Type
Equality comparison	==, !=	Boolean
Arithmetic comparison	<=, <, >=, >	Boolean
Bitwise operators	&, |, ^(exclusive OR), ~(negation)	Integer
Shift operators	«(left shift), »(right shift)	Integer
Unary operators	+, -	Integer
Pre- and postfix operators	++, --	Integer
Binary arithmetic operators	+, -, *, /, %, **(exponentiation)	Integer

Table 5.1: Integer Operators

5.1.2 Boolean

A value type **bool** (boolean) has only two possible values, true and false. Default value is false.

```
bool b = true
```

Operators

Operation	Operator Signs	Result Type
Equality comparison	==, !=	Boolean
Binary logical operators	&&, ||, !	Boolean

Table 5.2: Boolean Operators

The logical conjunction (&&) and the logical disjunction (||) have short-circuit behavior. If the left operand already reveals the answer, the right operand is not evaluated.

```
 // The second comparison (5>4) will not be evaluated
2 > 3 && 5 > 4

// The second comparison (5<4) will not be evaluated
2 < 3 || 5 < 4
```

5.1.3 Character

All ASCII characters are possible values. A character is enclosed in single quotes. Default value is '\0' (NULL).

```
char c = 'a'
```

Within a single-quoted character literal, the following escape codes can be used.

\0 ASCII null
\n new line
\' single quote
**** backslash

Backspace, carriage return, form feed, tab, vertical tab and Unicode code points (e.g. \u006A) are deliberately not supported.

Operators

Operation	Operator Signs	Result Type
Equality comparison	==, !=	Boolean
Relational comparison	<=, <, >=, >	Boolean

Table 5.3: Character Operators

5.1.4 Address

An address type holds a Bazo address of 256 bits (=32 byte). In the background it is nothing else than an alias for an integer represented in hexadecimal form.

```
address x = 0x2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
```

There are **no** member fields or functions for the address type, such as balance, send etc. In Lazo, they are defined as built-in functions. Please refer to section 6.7 for the built-in functions.

Operators

Address types support equality operators (==, !=) only.

5.1.5 Enum

In Lazo, enums store user-defined named constants and assigns them an integer value starting at 0. An enum type requires at least one member. An enum variable is default initialized with the first member.

```
// Enum declaration
enum Direction {
    NORTH,  // 0
    EAST,   // 1
    WEST,   // 2
    SOUTH   // 3
}

// usage
Direction d = Direction.NORTH
```

Operators

Operation	Operator Signs	Result Type
Equality comparison	==, !=	Boolean
Relational comparison	<=, <, >=, >	Boolean

Table 5.4: Enum Operators

5.2 Reference Types

Reference types are initialized with their corresponding default values. By doing so, there is no special handling required for the null pointer exceptions.

5.2.1 String

A String is an immutable sequence of characters. Strings are enclosed in double quotes and may contain arbitrary length of ASCII characters. Default value is "".

```
String text = "This is a string"
```

Within a double-quoted string literal, the following escape characters can be used.

\n newline
\" double quote
\\ backslash

Backspace, carriage return, form feed, tab, vertical tab and Unicode code points (e.g. \u006A) are deliberately not supported.

Operators

Operation	Operator Signs	Result Type
Equality comparison	==, !=	Boolean
Concatenation	+	String

Table 5.5: String Operators

Note that the concatenation works only with strings. Integer or boolean values should be explicitly converted to string in order to concatenate.

```
String s = "Number " + (String) 5       // "Number 5"
String s2 = "Boolean " + (String) true  // "Boolean true"
```

Member Functions

Function Signature	Description
int **length** ()	Returns the total number of characters in the string
char **at** (int index)	Returns the character at the given index
String **substr** (int start, int? length)	Extracts parts of the string. Length is optional. If omitted, it extracts the rest of the string.

Table 5.6: String Member Functions

5.2.2 Array

An array is a fixed-length sequence of the same type. By default, the elements are set to their default values when initialized. If not initialized, the default value is an empty array. Array

literal can be used to initialize an array with custom values, as shown below.

```
int[] nums = new int[4]              // initialized with default values
int[] nums2 = new int[]{0, 1, 2, 3} // initialized with an array literal
int[] nums3                          // empty array <=> new int[0]
```

Individual array elements can be accessed with square brackets. Array index usually starts at zero and ends with one less than the array length. Lazo also supports negative indexes which starts at -1 (last element) and goes backwards.

```
nums[0] = 1          // set a value
int x = nums[0]      // get the first value --> 1
int y = nums[-1]     // get the last value --> 0

int z = nums[13]     // ERROR: index 13 is out of bound
```

Operators

Array deliberately does not support any operators, not even equality operators (==, !=), because comparing the array's references do not make any sense in contracts.

Member Fields

- int **length**: returns the total array length

Iteration

Please refer to section 7.3.1 for how to iterate over an array.

Multi-dimensional Arrays

Lazo does not support multi-dimensional arrays.

5.2.3 Map

A map is an unordered collection of key-value pairs. The values can be accessed using the associated unique key. The default is an empty map without any keys. When accessing an undefined key, it will return the default value to avoid unnecessary exceptions. If needed, the contains() member function can be used to check the existence of the key in the map.

```
Map<String, int> map
map["one"] = 1

int x = map["one"]  // returns 1
int y = map["two"]  // returns 0 as default
delete map["one"]   // remove the entry with the key "one"
```

Unlike Solidity, a map entry can be deleted by the corresponding key in Lazo. Furthermore, Maps in Lazo are iterable, both the key and the value for each pair in the map are then returned. Please refer to section 7.3.2 about how iteration over a map works.

Member Functions

Function Signature	Description
bool **contains** (K key)	Checks if the key is in the map or not.

Table 5.7: Map Member Functions

5.2.4 Struct

Struct is an aggregate type which groups named variables into a single entity. Struct is normally a value type in languages such as C, C++, Java, C# etc. However, Struct in Lazo is a reference type just like in Solidity in order to save up storage / gas. By default, struct variables are initialized with the corresponding default values of the field. It is also possible to set custom values for specific fields using the named initialization.

```
// declaration
struct Person {
    address addr
    int balance
    bool available
}

Person p                                        // default initialization
Person p2 = new Person(0x123, 3, true)          // custom initialization
Person p3 = new Person(balance=3, address=0x123)    // named initialization

```

```
12 p.addr = 0x123       // set value
13 int x = p.balance    // get value
```

5.2.5 Error

Error type is similar to struct. Please refer to section 9.1 in chapter "Error Handling" for more information.

Chapter 6

Functions

Beside variables, functions are the most important concept in smart contract languages. But while solidity supports a whole range of function types, Lazo removes some of those to again simplify the language and make it more light-weight.

In this chapter, we will cover how Lazo supports functions and why certain functionality has been removed.

6.1 Visibility

Solidity supports four different kinds of visibilities: external, internal, public and private. Since we removed the inheritance from our language, Lazo only requires the following two visibilities:

- **Public** functions can be called from the current contract itself, but also from other contracts.
- **Internal** function can only be called from the current contract. They cannot be accessed by other contracts. However, because the contract is stored in the blockchain, those functions can be viewed by everyone.

The default visibility for functions is public (no keyword required). This can be changed by adding the **internal** keyword to the function declaration.

```
contract SimpleContract{
    // public function
    function String getName() {
        return "SimpleContract"
    }

    // internal function
    internal function String getInternalName() {
        return "InternalSimpleContract"
    }
}
```

6.2 Return Values

Functions can return values by specifying their return types in the function header. If a function does not return anything, void keyword is used. Lazo also supports multiple return values, as shown below.

```
contract SimpleContract{
    String name
    String version = "1.1"

    // No return value
    function void setName(String name) {
        this.name = name
    }

    // Single return value
    function String getName() {
        return name
    }

    // Multiple return values
    function (String, String) getContractInfos() {
        return name, version
    }
}
```

As too many return values make the code less readable, we limit the maximum amount of return values to three. Developers still have the possibility to wrap the return values using a struct.

In Solidity, there are two ways to return values: either with the return keyword followed by a comma-separated list of return values or with output parameters, which are specified in the function header by assigning a name to a return type as follows:

```
// Solidity >=0.4.16 <0.6.0
function arithmetic(uint _a, uint _b) public returns (uint o_sum, uint o_product)
{
        o_sum = _a + _b;
        o_product = _a * _b;
}
```

As this can be confusing and adds unnecessary complexity to the language, we decided to omit output parameters in Lazo.

6.3 Default & Named Parameters

Default parameters allow the function parameters to be initialized with default values when no value is passed. Thus, there is no function overloading required. Furthermore, Lazo also supports named parameters to initialize only certain default parameters as Python does.

```
function String greet(String greet, String name = "stranger", String title = "Mr.") {
    return greet + " " + title + " " + name
}

greet("Hello")                    // "Hello Mr. stranger"
greet("Welcome", title="Miss")    // "Welcome Miss stranger"
```

6.4 Constructor

As in Solidity, the constructor is executed only once during the contract creation. It cannot be invoked afterwards. The constructor function is used to initialize the contract. It can take parameters but cannot return any values. Therefore it does not support return types. Constructor is optional. There can be at most one constructor per contract.

```
version 1.0

contract SimpleContract{
    readonly int totalTokens
    Map<address, int> tokenHolders

    // constructor declaration
    constructor(int tokens) {
        totalTokens = tokens
        tokenHolders[msg.sender] = totalTokens
    }
}
```

6.5 Self-destruct

In the initial version of Lazo, there is no selfdestruct(address recipient) function available. In Solidity, this function is used to remove the contract code and storage from Ethereum blockchain. Furthermore, the remaining Ether will be sent to the designated recipient address.

Users can implement their own destruction mechanism using annotations for example.

6.6 Annotations

Instead of supporting function modifiers as Solidity does, we decided to use annotations to add additional behaviour to the functions. This makes the code more readable.

Annotations are enclosed between the opening square bracket symbol "[" and the closing square bracket symbol "]".

Table 6.1 shows all the supported annotations.

Annotation	Description
Pre	Precondition
Post	Postcondition
Payable	Allows the function to receive Bazo coins
Owner	Checks if the owner calls the function
ReadOnly	Function does not mutate the state of the contract
MaxCalls	Number of allowed calls within the same transaction (See 6.10)

Table 6.1: Function Annotations

```
contract SimpleContract{
    Map<address, int> balances

    [Payable]
    constructor() {
        balances[msg.sender] = msg.coins
    }

    // helper method
    internal function bool checkBalance(address account, int amount) {
        return balances[account] >= amount
    }

    // Annotation that checks amount and balances
    [Pre: amount > 0 && checkBalance(msg.sender, amount)]
    [Post: checkBalance(to, amount)]
    function void transfer(address to, int amount) {
        balances[msg.sender] -= amount
        balances[to] += amount
    }

    // Annotation that checks if the caller is the owner
    [Owner]
    function void changeOwner(address newOwner) {
        owner = newOwner
    }
}
```

6.6.1 Preconditions and Postconditions

Preconditions and postconditions are used to check certain conditions before and after executing a function. For example, developers can use it to verify if a user is allowed to execute this function or to check that the total is >= 0 after the function execution.

```
int total = 0

[Pre: number > 0]
[Post: total >= number]
function void countIfPositive(int number) {
    total += number
}
```

It is recommended to use an internal function if a condition is long or applied several times in the same contract.

Parameters specified in the function header can be passed to the annotations as well. As it may harm the reading flow, we initially considered placing these annotations between the function header and body. In the end, we decided to leave them where they are, as this would have a greater negative impact on the readability and would be an exception. Function-wide defined variables are available in the postcondition.

6.6.2 Payable

The Payable annotation is used to allow a function to receive Bazo coins sent with the transaction message. If the function is not marked so, it will reject the transaction message containing value.

```
[Payable]
function void deposit() {
    balances[msg.sender] += msg.coins
}
```

6.6.3 Owner

The Owner annotation is a shortcut for checking that the caller is the actual owner of the contract. When a contract is created, the owner field is automatically set to the contract creator (msg.sender). To change the owner, the function should be annotated with [Owner] as well, otherwise it throws a compile error.

```
[Owner]
function void changeOwner(address newOwner) {
    owner = newOwner
}
```

6.6.4 ReadOnly

ReadOnly marks the function as not mutating the state of the contract. If it does change the state, the compiler will throw an error. The following statements are considered as changing the state.

1. Change the state variable
2. Emit an event
3. Create a contract
4. Self destruct the contract
5. Send Bazo coins via calls
6. Call functions which are not marked as ReadOnly

When a ReadOnly function is called externally while not part of a transaction, it does not cost any gas since it does not change the state. However, it may cost gas if it is part of any transaction.

6.7 Global Built-in Functions

bool assert(bool condition)

This method is used to verify the program logic. If the condition is not met, it will throw an exception. Unlike Solidity, it will refund any remaining gas to the caller.

int balance (address account)

The balance method is used to query the balance of a specific address. This address can be the address of an Externally Owned Account or a Contract Account.

bool checkSig(String hash, string pubKeySig)

Verifies the signature in hash using the public key. The Bazo VM uses the *ecdsa.Verify*[3] function (Elliptic Curve Digital Signature Algorithm) of Go behind the scene.

int gasLeft()

Returns the gas that is left in the current function call.

void revert(string? message)

This method stops the actual execution and reverts back to the state that the contract had before the execution of the method. The message parameter is optional.

String sha3(String value)

It computes the hash value for the given value using the Secure Hash Algorithm 3 (SHA-3).

6.8 EVENTS

Solidity supports events which are used for simple logging facilities in Ethereum Virtual Machine (EVM). They can also be used to call JavaScript callback functions in Distributed Applications (DApps). These DApps can then listen to those events and handle them.

Lazo supports events as well. The VM currently does not support logging but might support it in the future.

```
1  version 1.0
2
3  contract ClientReceipt {
4      event Deposit(address _from, int _id, int _value)
5
6      [Payable]
7      function void deposit(int _id) {
8          emit Deposit(msg.sender, _id, msg.value)
9      }
10 }
```

When an event is called, its arguments are stored in the transaction's log. The logs are stored in the blockchain and are associated with the smart contracts address. Logs and events are not accessible from contracts.

Solidity supports the keyword *indexed*, which is allowed for up to three arguments and can be used to filter for specific values in the user interface of the DApp. Lazo omits this keyword as logging is not supported.

Potentially smart contracts could also listen for events in other smart contracts, but this would require **Account Abstraction**, which allows to store a certain amount of coins in the contract itself and enables the contract to pay for transactions himself. Both, Solidity and Lazo do not support this yet.

6.9 Recursion

Lazo does not support direct and indirect recursions within the same contract. They can be detected by the checker and will throw a compiler error.

Direct Recursion

```
function void directRecursion() {
    directRecursion() // Compiler Error
}
```

Indirect Recursion

```
function void a() {
    b()
}

function void b() {
    a() // Compiler Error
}
```

6.10 Cyclic Contract Calls

Cyclic contract calls are kind of indirect recursion but cannot be detected at compile-time.

To detect cyclic calls at run-time, the number allowed calls can be defined with the MaxCalls annotation, as shown below. As default, it is set to **one**. Every time the function is called, the counter will be decremented at run-time. If it reaches 0 and is called again, VM throws an exception.

```
[MaxCalls = 5]
function void callOtherContract() {
    MsgArgs m = new MsgArgs(70000)
    bank.withdraw().send(m)
}

// MaxCalls is one by default. Throws error if called again
function void callOtherContractOnce() {
    MsgArgs m = new MsgArgs(70000)
    bank.withdraw().send(m)
}
```

6.11 Lambda

Lambda functions are not supported in the first version of Lazo as they might not help with readability and we do not really see a great benefit from them as of now. Lambda support might be added in the future.

6.12 Fallback Function

Lazo does not support fallback function.

In Solidity, the fallback function is executed if no other functions match to the given function identifier and arguments. It is also executed whenever the contract receives Ether without any function call.

By default, the fallback function calls revert and throws an exception. If you want the contract to accept Ether, the fallback function should be implemented explicitly with the payable modifier. In a contract, only one fallback function is allowed and it should not have any arguments nor return values. Furthermore, it must be a public function.

```
// Solidity 0.4.25
function () payable { // no function name
    // empty block
}
```

Lazo omits the fallback function, because it could be confusing and misleading. A function should have one concrete goal. It should not be used for both receiving coins and executing default behavior in the same block. Most of all, if there is no suitable function found, the program should throw an error. It is wrong to handle the caller's error in a fallback function.

To receive funds, however, the contract can implement the payable interface. Please refer to section 10.3 Account Interface for an example.

Chapter 7

Control Structures

Other than Solidity, Lazo is not turing complete. This adds some constraints to the control structures as things like infinite loops are not supported. Due to this and for simplicity, features like while, goto, switch-case and do-while are not supported. Also for-loop is limited within a predefined range.

It is important to mention that the control structures in Lazo follow the same semantic rules as Solidity does.

7.1 If Statement

An if-statement is a control structure which enables alternative program paths during the run-time. An if-statement must have a conditional expression. The else if and else blocks are optional. Braces "{}" are mandatory, even if a block has only one statement.

```
if (x > 10) {
    // do something
} else if (x > 5) {
    // do something else if ...
} else {
    // do something else finally
}
```

7.1.1 Ternary Operator

Lazo supports the ternary operator as well. It is a shortcut for a simple if-else statement.

```
int absValue = (x < 0) ? -x : x
```

7.2 For Statement

A for-loop statement executes the statements in the loop body for the defined range. The range consists of start, stop and increment parameters. The loop is 0-index based. This means that by default the start value is 0 and it will be incremented by one up to the stop value, including this number.

Lazo allows only positive increment, i.e. the end and increment value must be positive numbers. There is however no explicit limit.

```
// from 0 to 10 increment by 1
for (x : to 10) {
    // x will be [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
}

// no index - from 0 to 10 increment by 1
for (_ : to 10) {
    // index is not needed
}

// from 1 to 10 increment by 1
for (x : 1 to 10) {
    // x will be [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
}

// from 1 to 10 increment by 2
for (x : 1 to 10 by 2) {
    // x will be [1, 3, 5, 7, 9]
}
```

7.2.1 Break

The break keyword can be used to exit the loop completely before the end. In the following example, the loop stops as soon as a prime number is found.

```
for (x : 101 to 200) {
    if (isPrime(x)) {
        break
    }
    // do something with non-prime number
}
```

7.2.2 Continue

The continue keyword can be used to skip the current iteration. In the following example, all the even indexes are skipped.

```
for (x : to 10) {
    if (x % 2 == 0) {
        continue
    }
    // do something odd
}
```

7.3 Foreach Statement

The foreach statement iterates over each element of a collection, such as an array or a map. It is important to note that modifying the collection during the iteration throws an exception.

The break and continue keywords are also possible in the foreach statements.

7.3.1 Iterate over an array

There are two ways to iterate over the array - with or without the index. The foreach statement executes the block for each element in the array, as shown below.

```
char[] vowels = new char[]{'a', 'e', 'i', 'o', 'u'}

foreach (char elem : vowels) {
    // do something with the element
}

// index is always int
foreach (index, char elem : vowels) {
    // do something with the index and element
}
```

7.3.2 Iterate over a map

When iterating over a map in other common languages, the order of the iteration is not specified and is not guaranteed to be the same as the insertion order.

In blockchain, however, the order should be guaranteed for deterministic result among all miners. Because of this reason, Lazo guarantees a consistent iteration order over map. Nevertheless, which iteration order (e.g. alphabetic or insertion order) the language implements is currently not defined. It will be decided at least when the Bazo VM is extended.

```
Map<address, int> balances
balances[0x2cf] = 50
// add more entries to the map

int total = 0
foreach (address account, int balance : balances) {
    // Do something with the account and/or the balance
    total += balance
}
```

Chapter 8

Expressions

An expression represents a computation of a new value from given values, variables, operators, and functions. The resulting new value can be a primitive data type (e.g. integer, boolean etc.) or a complex/composite data type (e.g. array, struct etc.).

8.1 Order of Evaluation

When evaluating such an expression, the language should avoid ambiguity and produce a unique parse tree. To resolve the ambiguities, Lazo use two additional rules:

- **Precedence Order**: Consider the expression 2+3*4. There are two possibilities: *(2+3)*4* or *2+(3*4)*. When two operators share an operand, the operator with higher precedence is resolved first. Since * has higher precedence than +, the expression is evaluated as 2+(3*4).
- **Associativity**: When an expression has operators with the same precedence, its associativity rule is applied. For example, addition 1+2+3 is evaluated with left associativity (1+2)+3. On the other hand, exponentiation 5**2**2 is evaluated with right associativity, namely 5**(2**2).

The following table lists all the operators in Lazo with their precedence level and associativity [6][8]. The table is order from highest precedence to lowest precedence.

Precedence	Operator	Description	Associativity
1	x++	postfix increment	left to right
	x--	postfix decrement	
	<array>[index]	array element access	
	<object>.<member>	object member access	
	(<expression>)	parentheses	
	<func>(<args>)	function call	
	new <type>	object creation	
2	++x	prefix increment	right to left
	--x	prefix decrement	
	+x	unary plus	
	-x	unary minus	
	!x	logical NOT	
	~x	bitwise NOT	
3	(type) x	type casting	right to left
4	5 ** 2	exponentiation	right to left
5	2 * 3	multiplication	left to right
	10 / 2	division	
	5 % 2	modulo	
6	3 + 4	addition	left to right
	4 - 3	substraction	
	"hello" + "world"	string concatenation	
7	2 « 3	bitwise left shift	left to right
	2 » 3	bitwise right shift	
8	2 < 3	less than	left to right
	2 <= 3	less than or equal	
	3 > 2	greater than	
	3 >= 2	greater than or equal	
9	x == 2	equality	left to right
	x != 2	inequality	
10	5 & 3	bitwise AND	left to right
11	5 ^ 3	bitwise XOR	left to right
12	5 | 3	bitwise OR	left to right
13	true && true	logical AND	left to right

14	true || false	logical OR	left to right
15	<cond.> ? <exp> : <exp>	ternary operator	right to left
16	x += 2	shorthand assignment	right to left
	x -= 2		
	x **= 2		
	x *= 2		
	x /= 2		
	x %= 2		
	x «= 2		
	x »= 2		
	x &= 2		
	x ^= 2		
	x |= 2		
17	<exp> , <exp>	comma, sequence	left to right

Table 8.1: Operator Precedence and Associativity

It is important to note that in Solidity, the bitwise AND, OR and XOR operators have higher precedence than the relational (<, <=, >, >=) and equality (==, !=) operators. But it is vice versa in other common popular languages, such as Java, JavaScript, C# etc. Lazo also follows the same precedence order of the other common languages to keep consistency.

Chapter 9

Error Handling

Solidity's error handling is quite limited. Solidity has three different functions - assert, require and revert. *Revert* reverts the state. *Assert* is used to check for errors in the contract's logic and *require* is used for user input checks. In all three cases, solidity throws an exception which bubbles up and notifies the caller about the error. Important: **The state is always reverted!**

Unfortunately, Solidity does not allow to catch and handle exceptions as a programmer usually does in other programming languages. As we were not sure if this is the best approach to follow, we analyzed different programming languages. While Vyper[7] and Scilla[5] both do not support error handling, other common languages such as Golang, Rust or C use different approaches:

- C: Returning specific error values as int
- Go: multiple return values and error types
- Rust: passing error structs into the function or monadic error handling.

During our analysis, we found that only two approaches are widely used in modern programming languages[4], namely throw exceptions and return error values. Single integer return value could be misleading, because the error code could be also a possible value. Therefore we decided to focus on a solution using either throw-catch exceptions or multiple return values with error types as Go does.

We recognized that Go returns nil if no error has occurred. However, our language does not support null values due to simplicity. From our findings, we decided that we will continue with the simple and straightforward approach of exceptions and add the possibility to catch exceptions as well. But in contrast to other languages we will call them **errors** instead of

exceptions as this is more precise.

9.1 Error Declaration and Usage

Custom error types can be declared with the error type keyword. The syntax is similar to struct declaration.

```
// without any fields
error MyError { }

// with fields
error MyDetailedError {
    int code
    string message
}
```

When an error has occurred, the error can be thrown with an appropriate error type as follows.

```
throw MyError{}

throw MyDetailedError{100, "An error has occured"}
```

The occurred errors bubble up through the call stack and cannot be caught or handled. It means the transaction will always be reverted if an error has occurred.

We considered catching an error and let the developers to define an alternative program logic in the catch-block. However, we did not find any useful use cases for that. A transaction should be atomic. It succeeds as whole or nothing occurs. Therefore, catching error is not supported and the state will always be reverted. This might change in the future.

9.2 Built-in Errors

ArgumentError {string message}
: Provided argument is invalid

ArithmeticError {string message}

Arithmetic error has occurred, e.g. division by zero.

ArrayIndexOutOfBoundError {int index, int length}

Array index is out of range.

Error {string message}

A generic error type which can be used to throw any error.

NoSuchFunctionError

The required function is not available or does not match with the signature.

NotPayableError

The function or contract is not payable. For function, add the [Payable] annotation. For contract, implement the Payable interface.

OutOfGasError

Allocated gas is used up. Not enough gas to complete the transaction.

Chapter 10

Polymorphism

10.1 Contracts

Inheritance is not supported in Lazo. The reason for this is that inheritance "hides" some implementation in the super class. This can have a negative impact on the auditability.

10.1.1 *new*-Keyword

In contrast to Solidity, Lazo does not support instantiating new contracts within a contract. Therefore the new-keyword is omitted for contracts. The reason for this is that this feature is very rarely used in smart contracts and therefore leads to unnecessary complexity. As an alternative, it is recommend to use interfaces.

10.2 Interfaces

Interfaces are supported by Lazo as they are very useful for the following two use cases:

- Adapt to templates (e.g. ERC20 Tokens)
- Simplify calling methods on other contracts.

In addition to that, using interfaces helps creating more readable code.

10.2.1 Adapt to Templates

Interface is like a template. It declares only the function signatures (name, arguments and return types). The implementation is done in the contract. By using the is-keyword in the contract declaration, the compiler checks whether the contract implements all the methods defined in the interface. If not, a compiler error is thrown.

```
interface ERC20 {
    // Definition of the interface
}

contract MyContract is ERC20 {
    // Implementation of the interface and additional functionality
}
```

10.2.2 Simplify Calling Methods on Other Contracts

Calling functions on other contracts is not very simple. If you want to call a method on another contract, you need to know the first four bytes of the hash of the function name, which is very low-level and not human-friendly at all. To solve this problem, interfaces can be used.

First, the developer creates an interface for the desired contract. Then, he can cast the contract's address to the interface and call the methods on it as usual.

Note that the external method call is not executed as long as the send() function has not been called.

bool send (MsgArgs? args)

This method is used to send additional arguments with the transaction, such as gas, coins etc. If none set, the method execution is limited to a default gas value.

struct MsgArgs{int gas; int coins}

This struct is used to send meta data (e.g. gas limit and coins) with the function call. The data can be then accessed via the global variable (See 4.8.1).

```
interface Bank {
    [Payable]
    void deposit()

    void withdraw(int amount)
}

contract MyContract {

    // Cast the address of the other contract instance to the interface type
    Bank bank = (Bank) 0x12345...

    function void myDeposit() {
        // Use the interface to call the deposit function in the other contract
        MsgArgs args = new MsgArgs(coins=10, gas=21000)
        bank.deposit().send(args)
    }

    function void myWithdraw(int amount) {
        // Use the interface to call the withdraw function in the other contract
        bank.withdraw(amount).send()
    }
}
```

10.3 Payable Interface

Lazo provides a built-in Payable interfaces to send coins to externally owned accounts or contract accounts. Figure 13.2 shows the available functions.

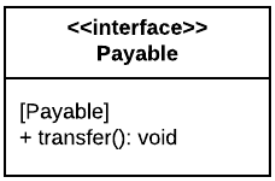

Figure 10.1: Payable Interface

10.3.1 Send coins to an Externally Owned Account

To send coins to an externally owned account, use the transfer() function, as shown below.

```
// Address of an externally owned account
Payable person = (Payable) 0x12345...

// Send coins to a EOA
MsgArgs args = new MsgArgs(coins=10)
person.transfer().send(args)
```

10.3.2 Send coins to a Contract Account

To send coins to a contract account, transfer() function can be used as well. However, it is important that the contract should have implemented the Payable interface. Otherwise, it throws a **NotPayableErrror** and the coins will be refunded.

```
// Payable implementation
// ----------------------
contract Bank is Payable {
    Map<address, int> balances

    // Not all variables/functions are shown here

    [Payable]
    function void transfer() {
       balances[msg.sender] += msg.value
    }
}

// Usage
// -----
// Address of a payable contract
Payable bank = (Payable) 0x12345...

// Send coins to a contract
MsgArgs args = new MsgArgs(coins=10, gas=30000)
bank.transfer().send(args)
```

Chapter 11

Proposals

11.1 Account Abstraction

Account Abstraction is used to enable contracts to store coins and be able to pay for transactions by themselves. Account Abstraction most likely requires a new kind of transaction which must be added to the Bazo VM. This could be a helpful feature for the future and might be designed and implemented in another project.

11.2 Account Reference

Account References could be used to associate tokens or coins with references instead of with addresses within the contract. In the case of an exchange site, the contract would store the reference to the customer instead of the address of the exchange site. The customer could then also purchase further funds and associate them with the same reference.

While this could be a useful feature, especially for Initial Coin Offerings (ICOs), there are still some security concerns. Therefore, Lazo does not provide any language features to support it. Developers could implement it using the reference as a parameter and adding security checks using the precondition annotations. We will not go into further detail since it is not part of this study work.

Part III

Implementation

Chapter 12

Lazo Grammar in ANTLR

12.1 Lexer Rules

```
// Reserved Keywords (Hint: Order by asc)
// ------------------
BREAK: 'break' ;
BY: 'by' ;
CONSTRUCTOR: 'constructor' ;
CONTINUE: 'continue' ;
CONTRACT: 'contract' ;
ELSE: 'else' ;
EMIT: 'emit' ;
ENUM: 'enum' ;
EVENT: 'event' ;
FOR: 'for' ;
FOREACH: 'foreach' ;
FUNCTION: 'function' ;
INTERFACE: 'interface' ;
INTERNAL: 'internal' ;
IF: 'if' ;
IS: 'is' ;
MAP: 'Map' ;
READONLY: 'readonly' ;
RETURN: 'return' ;
STRUCT: 'struct' ;
THROW: 'throw';
```

```
24  TO: 'to' ;
25  VERSION: 'version' ;
26
27  BOOL
28    : 'true'
29    | 'false'
30    ;
31
32  // Reserved Keywords which are not used in the language (prohibited) - Hint: Order by asc
33  // ----------------
34  ABSTRACT: 'abstract' ;
35  AS: 'as' ;
36  CASE: 'case' ;
37  CATCH: 'catch' ;
38  CONST: 'const' ;
39  EXTENDS: 'extends' ;
40  EXTERNAL: 'external' ;
41  FINALLY: 'finally' ;
42  GOTO: 'goto' ;
43  IMPLEMENTS: 'implements' ;
44  NULL: 'null';
45  OUT: 'out' ;
46  OVERRIDE: 'override' ;
47  PRIVATE: 'private' ;
48  PUBLIC: 'public' ;
49  REF: 'ref' ;
50  STATIC: 'static' ;
51  SWITCH: 'switch' ;
52  TRY: 'try' ;
53  VAR: 'var' ;
54  VIRTUAL: 'virtual' ;
55  WHILE: 'while' ;
56  // ----------------
57
58  // Punctuation marks
59  // ----------------
60  LPAREN: '(' ;
61  RPAREN: ')' ;
62  LBRACE: '{' ;
63  RBRACE: '}' ;
64  LBRACK: '[' ;
65  RBRACK: ']' ;
```

```
66  SEMI: ';';
67  COMMA: ',';
68  DOT: '.';
69  // -----
70
71  // Arithmatics
72  // -----------
73  PLUS: '+';
74  MIN: '-';
75  MUL: '*';
76  DIV: '/';
77  MOD: '%';
78  EXP: '**';
79  LSHIFT: '<<';
80  RSHIFT: '>>';
81
82  // Logical Operators
83  AND: '&&';
84  OR: '||';
85  NOT: '!';
86  BITWISE_AND: '&';
87  BITWISE_OR: '|';
88  CARET: '^';
89  TILDE: [\u007e];
90
91  // Comparison
92  // ---------
93  EQ: '==';
94  NEQ: '!=';
95  GT: '>';
96  GT_EQ: '>=';
97  LT: '<';
98  LT_EQ: '<=';
99  // ---------
100
101 IDENTIFIER
102   : ( '_' | ALPHA_LETTER) ( '_' | ALPHA_LETTER | DEC_DIGIT )* ;
103
104 fragment ALPHA_LETTER
105   : [a-zA-Z] ;
106
107 INTEGER
```

```
108     : DEC_DIGIT_LIT
109     | HEX_DIGIT_LIT ;
110
111   HEX_DIGIT_LIT
112     : '0x' HEX_DIGIT+ ;
113
114   fragment HEX_DIGIT
115     : [0-9a-fA-F] ;
116
117   DEC_DIGIT_LIT
118     : DEC_DIGIT+ ;
119
120   fragment DEC_DIGIT
121     : [0-9] ;
122
123   STRING
124     : '"' UNICODE_CHAR* '"' ;
125
126   CHARACTER
127     : '\'' ( ESCAPED_CHAR | UNICODE_CHAR ) '\'' ;
128
129   fragment ESCAPED_CHAR
130     : '\\' ( '0' | 'n' | '\\' | '\'' | '"' ) ;
131
132   fragment UNICODE_CHAR
133     : ~[\r\n] // any Unicode code point except carrige return & new line
134     ;
135
136   NLS
137     : NL+;
138
139   fragment NL
140     : [\n]
141     | [\r\n]
142     ;
143
144   // Skip Rules
145   // --------
146   WHITE_SPACE
147     : [ \t\f\r]+ -> skip // skip spaces, tabs, form feed and carrige return
148     ;
149
```

```
LINE_COMMENT
  : '//' ~[\r\n]* -> skip ;

BLOCK_COMMENT
  : '/*' .*? '*/' -> skip ;
```

12.2 Parser Rules

```
program
  : NLS* versionDirective interfaceDeclaration* contractDeclaration EOF ;

versionDirective
  : 'version' INTEGER '.' INTEGER NLS ;

interfaceDeclaration
  : 'interface' IDENTIFIER '{' NLS* interfacePart* '}' NLS ;

interfacePart
  : functionSignature NLS ;

functionSignature
  : annotation* ( type | '(' type (',' type)* ')' ) IDENTIFIER '(' paramList? ')' ;

contractDeclaration
  : 'contract' IDENTIFIER ('is' IDENTIFIER (',' IDENTIFIER)* )?
    '{' (NLS | contractPart)* '}' NLS? ;

contractPart
  : variableDeclaration
  | structDeclaration
  | errorDeclaration
  | enumDeclaration
  | eventDeclaration
  | constructorDeclaration
  | functionDeclaration
  ;

// Declarations
```

```
31  // ------------
32
33  variableDeclaration
34    :  'readonly'? type IDENTIFIER assignment? NLS;
35
36  structDeclaration
37    : 'struct' IDENTIFIER '{' NLS* structField* '}' NLS ;
38
39  errorDeclaration
40    : 'error' IDENTIFIER '{' NLS* structField* '}' NLS ;
41
42  structField
43    : type IDENTIFIER NLS;
44
45  eventDeclaration
46    : 'event' IDENTIFIER '(' paramList? ')' NLS;
47
48  enumDeclaration
49    : 'enum' IDENTIFIER '{' NLS* IDENTIFIER (',' NLS* IDENTIFIER)* NLS*'}' NLS ;
50
51  constructorDeclaration
52    :  annotation* 'constructor' '(' paramList? ')' statementBlock ;
53
54  functionDeclaration
55    : annotation* functionHead statementBlock ;
56
57  functionHead
58    : 'internal'? 'function' (type | '(' type (',' type)*')') IDENTIFIER '(' paramList? ')' ;
59
60  annotation
61    : '[' IDENTIFIER (':' expression)? ']' NLS ;
62
63  paramList
64    : parameter (',' parameter)* (',' defaultParameter)* ; // todo allow optional newline
65
66  parameter
67    : type IDENTIFIER ;
68
69  defaultParameter
70    : parameter assignment ;
71
72  // Types
```

```
73  // -----
74
75  type
76    : arrayType
77    | mapType
78    | IDENTIFIER ;
79
80  arrayType
81    : IDENTIFIER '[' ']' ;
82
83  mapType
84    : 'Map' '<' type ',' type '>' ;
85
86  // Statements
87  // ----------
88
89  statementBlock
90    : '{' ( NLS | statement )* '}';
91
92  statement
93    : assignmentStatement
94    | returnStatement
95    | expressionStatement
96    | sendStatement
97    | emitStatement
98    | variableDeclaration
99    | ifStatement
100   | forEachStatement
101   | forStatement
102   | mapForEachStatement
103   | breakStatement
104   | continueStatement
105   | throwStatement
106   ;
107
108 emitStatement
109   : 'emit' expression NLS ;
110
111 deleteStatement
112   : 'delete' expression NLS ;
113
114 ifStatement
```

```
115     : 'if' '(' expression ')' statementBlock
116       ('else if' '(' expression ')' statementBlock)?
117       ('else' statementBlock)? ;
118
119   forStatement
120     : 'for' '(' IDENTIFIER ':' rangeStatement ')' statementBlock ;
121
122   forEachStatement
123     : 'foreach' '(' (IDENTIFIER ',')? type IDENTIFIER ':' expression ')' statementBlock ;
124
125   mapForEachStatement
126     : 'foreach' '(' type IDENTIFIER ',' type IDENTIFIER ':' expression ')' statementBlock ;
127
128   breakStatement
129     : 'break' NLS ;
130
131   continueStatement
132     : 'continue' NLS ;
133
134   rangeStatement
135     : expression? 'to' expression ('by' expression)? ;
136
137   expressionStatement
138     : expression NLS ;
139
140   sendStatement
141     : expression '.' 'send' '(' expression? ')' NLS ;
142
143   argumentList
144     : expression (',' expression)* (',' namedArgument)*
145     | namedArgument (',' namedArgument)*
146     ;
147
148   namedArgument
149     : IDENTIFIER '=' expression ;
150
151   assignmentStatement
152     : expression assignment NLS ;
153
154   assignment
155     : '=' expression ;
156
```

```
designator
  : IDENTIFIER ;

throwStatement
  : 'throw' IDENTIFIER '{' argumentList? '}' NLS ;

returnStatement
  : 'return' (expression (',' expression)*)? NLS ;

// Expressions
// -----------
expression
  : expression ( '++' | '--' )
  | expression '[' expression ']'    // index access
  | expression '.' IDENTIFIER        // member access
  | expression '(' argumentList? ')' // call
  | newCreation
  | '(' expression ')'
  // --- End of Level 1 ----
  | <assoc=right> ( '++' | '--' | '+' | '-' | '!' | TILDE ) expression
  | <assoc=right> '(' type ')' expression      // cast
  | <assoc=right> expression '**' expression
  | expression ('*' | '/' | '%') expression
  | expression ('+' | '-') expression
  | expression ('<<' | '>>') expression
  | expression ('<' | '>' | '<=' | '>=') expression
  | expression ('==' | '!=') expression
  | expression '&' expression
  | expression '^' expression
  | expression '|' expression
  | expression '&&' expression
  | expression '||' expression
  | <assoc=right> expression '?' expression ':' expression
  | <assoc=right> expression ( '+' | '-' | '**' | '*' | '/' | '%'
                      | '<<' | '>>' | '&' | '^' | '|' ) '=' expression
  | operand
  ;

newCreation
  : structCreation
  | arrayCreation
```

```
198     | mapCreation
199     ;
200
201 structCreation
202     : 'new' IDENTIFIER '(' argumentList? ')' ;
203
204 arrayCreation
205     : 'new' IDENTIFIER ('[' expression ']' ('{' '}')?
206             | '[' ']' '{' expression (',' expression)* '}') ;
207
208 mapCreation
209     : 'new' mapType '(' ')';
210
211 operand
212     : literal
213     | designator
214     ;
215
216 literal
217     : INTEGER
218     | CHARACTER
219     | STRING
220     | BOOL
221     ;
```

Chapter 13

Syntax Verification

The syntax of the Lazo language is extensively tested with ANTLR and Java. Depending on the language feature, different testing methods have been applied, as follows:

- Lexer
    1. **Lexemes**: A stream of characters was read in and tokenized (e.g. integer, string etc.). The produced tokens were checked whether their type and their content were correct.
    2. **Features**: Language features (e.g. struct, function etc.) were tokenized and verified whether a long stream of characters were splitted into tokens correctly.
- Parser
    1. **Nodes**: A stream of characters was tokenized and parsed into an abstract syntax tree with nodes, such as *Program, Contract, Function, Expression* etc. The nodes were checked whether they had correct tokens without any errors.
    2. **Contracts**: Complete valid contract examples (e.g. Purchase, OpenAuction, BlindAuction etc.) were parsed and verified that no errors had occurred.

In general, **all the defined features** in the specification are verified with little code snippets and full complete contract examples.

13.1 Test Summary

117	0	0	0.530s
tests	failures	ignored	duration

100% successful

Packages Classes

Package	Tests	Failures	Ignored	Duration	Success rate
lexer.features	8	0	0	0.130s	100%
lexer.lexemes	10	0	0	0.018s	100%
parser.contracts	5	0	0	0.194s	100%
parser.nodes	94	0	0	0.188s	100%

Figure 13.1: Test Summary

Packages Classes

Class	Tests	Failures	Ignored	Duration	Success rate
lexer.features.TestAnnotations	1	0	0	0.066s	100%
lexer.features.TestEnum	1	0	0	0.008s	100%
lexer.features.TestEvents	1	0	0	0.008s	100%
lexer.features.TestFunctions	1	0	0	0.011s	100%
lexer.features.TestInterfaces	1	0	0	0.007s	100%
lexer.features.TestNamedParams	1	0	0	0.010s	100%
lexer.features.TestSimpleContract	1	0	0	0.010s	100%
lexer.features.TestStruct	1	0	0	0.010s	100%
lexer.lexemes.TestCharacter	1	0	0	0.008s	100%
lexer.lexemes.TestFixTokens	2	0	0	0.001s	100%
lexer.lexemes.TestIdentifier	3	0	0	0s	100%
lexer.lexemes.TestInteger	3	0	0	0.002s	100%
lexer.lexemes.TestString	1	0	0	0.007s	100%
parser.contracts.TestExamples	5	0	0	0.194s	100%
parser.nodes.TestContract	4	0	0	0.024s	100%
parser.nodes.TestExpression	25	0	0	0.037s	100%
parser.nodes.TestFunctions	7	0	0	0.063s	100%
parser.nodes.TestInterface	3	0	0	0.012s	100%
parser.nodes.TestStatements	36	0	0	0.044s	100%
parser.nodes.TestTypes	19	0	0	0.008s	100%

Figure 13.2: Test Details

Part IV

Evaluation

Chapter 14

Results

14.1 Achievements

14.1.1 Rough Analysis

In the rough analysis, 24 existing smart contract languages are collected and roughly analyzed to identify the key characteristics of a language for the blockchain. See Comparison of Smart Contract Languages for complete overview of the analysis.

As a result, it seems to be that smart contract languages are predominantly imperative and statically typed, as they are more straightforward and easier to understand. By enforcing static types, many programming errors could be detected at compile-time before deploying the contracts to the blockchain. Furthermore, some languages support object orientation, however inheritance is not encouraged. It is important to mention that turing-completeness is a major topic of debate among smart contract languages. A little more than half of the analyzed languages are not turing-complete because programming a endless loop in a contract is not desired.

14.1.2 Domain Analysis

In the detailed analysis, the following well elaborated smart contract languages were analyzed in great detail:

- **Solidity** is the most popular and widely used smart contract language for Ethereum blockchain. Since it is a turing-complete language, basically any computer program could be written in Solidity. Yet, Solidity is overloaded with too many features and is therefore quite complex.
- **Vyper** is also for Ethereum blockchain, however it removes the complexity of Solidity and makes the language easier to understand. It also addresses the problems Solidity has and provides alternative solutions. By analyzing Vyper, those problems could be eliminated in Lazo as well.
- **Scilla** is a functional programming language for Zilliqa blockchain. In comparison to other languages, Scilla follows different approaches for programming contracts. For instance, it uses the *continuation-passing style* and clearly separates the in-contract computation and communication between contracts at language level. Even though it is an interesting approach, it adds more complexity to the language. After some analysis and discussions, the approaches of Scilla were not considered in Lazo.

During the analysis, their features, syntax and contract examples were documented. Furthermore, famous or frequent attacks on Solidity were also analyzed (e.g. re-entrancy attack, integer overflow and underflow, gas limit and loops etc.). See the domain analysis document for complete results.

14.1.3 Language Design

With the acquired knowledge about smart contracts, Lazo language was specified in an agile manner. As a result, Lazo is designed to be a statically typed, imperative and non-turing complete programming language. Thus, Lazo is easier to understand and more robust against errors. Even though Lazo is inspired by Solidity, many unnecessary features are removed and essential features are simplified when needed. By doing so, Lazo has become much more simple.

All language features are documented with illustrative code snippets. The Lazo grammar is also written in ANTLR and thoroughly verified with Java. Before writing the ANTLR grammar, a first version of Extended Backus-Naur Form (EBNF) was created. Unfortunately, there were no tools to verify the EBNF grammar. However, there was a tool to generate EBNF from the ANTLR grammar. Since the ANTLR grammar could be verified for correctness, we omitted improving the first version of EBNF. Once the ANTLR grammar was completed, EBNF was

automatically generated from it.

Initially, Lazo was designed with semicolons ";" as statement separators. Later on it was redesigned with newlines to make the language more readable. However, the parser rules have become more complex because of that. Every newline in the source code is part of the abstract syntax tree now and each parser rule should handle it. In some cases, a rule requires at least one newline but in most cases newlines could be ignored.

Security concerns are also taken into account and countermeasures are built-in at language level, where possible.

- **Re-entrancy attack**: Cyclic external contract calls are prevented using the function annotation MaxCalls, which limits the number of function calls in the same call stack.
- **Gas limit and loops**: Lazo does not allow loops over a variable. Therefore, there is no endless loop possible. The point of termination and the cost of gas could be calculated precisely.
- **Integer overflow and underflow**: Lazo supports only big integer, which indeed takes care about the overflow and underflow problem.
- **Contract ownership**: Lazo supports Owner function annotation, which guarantees that only the contract owner can call certain functions. When changing the ownership, Lazo also checks that only the owner can do so.

Furthermore, contract examples from Solidity are translated to Lazo in order to prove that the real-world use cases can be programmed with Lazo as well.

14.2 Not Achieved Goals

14.2.1 Syntax verification with Go

Initially, it was planned to test the ANTLR grammar with Go. However, the ANTLR tool to generate Go files had some bugs during this study work (See the StackOverflow question *53100633*). Since the project has limited time period, the lexer and parser for the ANTLR grammar were generated in Java and tested with JUnit library.

14.2.2 Checker Rules

At the start of the project, two weeks were allocated to specify the checker rules within this study work. However, in the middle of the project we had to redesign our language syntax with newlines. It took a considerable amount of time to modify the already written ANTLR grammar, verify the syntax and to update all the code snippets in the document. Apart from that, we also received a new task to present our language at the Bazo workshop towards the end of the project. Because of these reasons, the specification of the checker rules is postponed to our bachelor thesis.

Chapter 15

Conclusion

Summary The goal of this study work was to specify a smart contract language for the Bazo blockchain. Nevertheless, exact requirements for that language were not known at the beginning. With the acquired knowledge from the analysis phases and the consultation of our supervisor, the Lazo language could be designed. It took many iterations and discussions to get the language syntax right. Eventually, Lazo has got its shape and made possible to read and write smart contracts easier at the high-level language.

Unique Features Even though Lazo is inspired by Solidity, Lazo has many unique features and outstands other smart contract languages, such as statement separation by a newline, function annotations for checking conditions, foreach-loop with access to the current index, comprehensible interfaces to send coins and call external functions etc. Most importantly, there will be no null pointer exceptions because in Lazo all variables are default initialized. Due to that, a lot of null checks could be spared. In conclusion, Lazo is a new kind of approach for creating smart contracts on the blockchain.

Suggestion for improvements We did not have a lot of experience with blockchain and contract programming. When designing a feature, the underlying processes of the Bazo blockchain were sometimes not clear to us. Therefore, we considered Bazo blockchain as a black box and designed the language on top of that. If we had had more clear internal view, we would have been able to design even a better language. Apart from that, lack of contract programming raised numerous questions like: "do we really need this feature, is there a real-world use case for this feature or is it the recommended best practice to program?". It eventually cost us more time. With prior knowledge, we could have worked efficiently without redesigning certain features several times (e.g. error handling, transferring coins etc.).

Future Work According to the language specification, a **compiler** could be developed to compile Lazo programs into Bazo virtual machine instructions (opcodes). If there are no opcodes available for certain new features, the Bazo VM needs to be extended. In addition to that, an **IDE extension** for syntax highlighting and code completion would be also very convenient for writing contracts in Lazo.

Glossary

abstract syntax tree is a tree representation of the abstract syntactic structure of source code written in a programming language.. 75

call stack A stack data structure which stores data about the active subroutines of a program. 80

Externally Owned Account is controlled by private keys and has no associated code. 8, 45, 62

Hoisting Hoisting is JavaScript's default behavior of moving declarations to the top.. 25

Lazo Lazo is the name of our programming language. 18–22, 25–28, 30, 32, 33, 35, 36, 38–41, 45–50, 52, 53, 55, 59, 61, 63

Acronyms

ANTLR ANother Tool for Language Recognition. 12, 75, 79

ASCII American Standard Code for Information Interchange. 19, 32, 33

DApp Distributed Application. 45, 46

EBNF Extended Backus-Naur Form. 79

EVM Ethereum Virtual Machine. 45

ICO Initial Coin Offering. 63

IL Intermediate Language. 18

Opcodes Operation codes. 11, 18

SI Internation System of Units. 29

VM Virtual Machine. 18, 30

Bibliography

[1] Marco Steiner Ennio Meier. *Bachelor Thesis - Integrating Smart Contracts into the Bazo Blockchain*. 2018.

[2] *Bazo Blockchain Github Repo*. URL: https://github.com/bazo-blockchain. (accessed: 7.12.2018).

[3] Golang Docs. *Package ecdsa*. URL: https://golang.org/pkg/crypto/ecdsa/#Verify. (accessed: 28.10.2018).

[4] Danny van Heumen. *Error handling in modern languages*. URL: http://dannyvanheumen.nl/post/error-handling-in-modern-languages/. (accessed: 24.10.2018).

[5] Scilla. *Scilla Lang*. URL: https://scilla-lang.org/. (accessed: 30.09.2018).

[6] Solidity Docs v0.5.0. *Order of Precedence of Operators*. URL: https://solidity.readthedocs.io/en/latest/miscellaneous.html#order. (accessed: 28.10.2018).

[7] Vyper. *Vyper 0.1.0-beta3 documentation*. URL: https://vyper.readthedocs.io/en/latest/index.html. (accessed: 26.09.2018).

[8] Robert Sedgewick & Kevin Wayne. *A Operator Precedence in Java*. URL: https://introcs.cs.princeton.edu/java/11precedence/. (accessed: 22.10.2018).

Part V

Appendices

Appendix A

Rough Analysis

Google Sheet: Comparison of smart contract languages

Language	Blockchain	Quality of Sources (1 non-reliable , 10 very reliable)	Popularity in GitHub (1 unpopular 10 very popular > 3000)	Quality of Lang Specs (1 = not at all, 10 = in great detail)	Imperative	Functional	Object-oriented	Inheritance
Ivy	Bitcoin	7	5	4	X		X	
Simplicity	Bitcoin	5	1	5		X		
Balzac	Bitcoin	3	1	3		X		
Solidity	Ethereum	10	10	10	X		X	X
Vyper	Ethereum	8	8	8	X		X	
Bamboo	Ethereum	3	4	2	X		X	
Flint	Ethereum	9	3	8	X		X	X
Idris	Ethereum	8	8	9		X	modular	
L4	Ethereum	1	1	1	??	??	??	??
Babbage	Ethereum	2	N/A	1				
Lolisa	Ethereum	3	N/A	2	X		X	X
Serpent	Ethereum	3	3	5	X		X	X
eWASM (WebAs	Ethereum	3	3	2	X			
Mutan	Ethereum	3	1	1	X			
LLL	Ethereum	4	N/A	1	X	X		
Sophia	Eternity	4	2	4		X	X	X
Varna	Eternity	2	N/A	2	X		X	
Scilla	Zilliqa	9	2	6		X		
Michelson	Tezos	2	N/A	3	X			
Liquidity	Tezos	4	2	3		X		
Pact	Kadena	4	3	3	X		modular	
Plutus	Cardano	7	2	7		X	data types (haskell)	X
Marlowe	Cardano	2	1	1	??	??	??	??
RIDE	Waves	4	2	8		X	X	
F*	Zen	5	5	6	X	X	X	
Rholang	RChain	5	2	4	X		X	
Lisk	Ethereum	8	8	1	??	??	??	??

Modifiers	Procedural	Static types	Dynamic types	Turing Complete	Intermediate Language	Active
		X				X
		X				??
		X				X
X		X		X		X
		X				X
		X				4m ago
X		X				X
X		X		X		X
??	??	??	??	??	??	??
				??		??
X		X		X		??
	X		X	X		Deprecated
		X		X	X	X
	X			X		Deprecated
	X		X	X		??
X		X		X		X
X		X				??
		X			X	X
	X	X				
		X				X
	X	X				X
	X	X		X		X
??	??	??	??	??	??	??
		X				X
		X		Prooves termination		X
		X				X
??	??	??	??	??	??	??

Bemerkung	Link
compiles to Bitcoin Script, so not turing-complete and very limited	https://github.com/ivy-lang/ivy-bitcoin
	simplicity smart c…
Abstract language/Sieht nicht wirklich nach smart contracts aus, sehr sehr basic und kompliziert	
	https://github.com/ethereum/solidity
Experimental language. Syntax similar to Python3, but not all Python3 fur	https://github.com/ethereum/vyper
	https://github.con https://medium.com/@pirapira/implementing-a-vault-in-bamboo-9c08241b6755
Inspired by Swift, still in alpha development	https://github.com/flintlang/flint
general-purpose functional programming language with dependent types,	https://github.com/idris-lang/Idris-dev
No information available	
Visual/Mechanical langauge	
Largely subset of Solidity, programs written in Solidity can be translated ir	https://arxiv.org/abs/1803.09885
Python-like	
Ethereum flavored WebAssembly, still under development	
Lisp-like	
ML (MetaLanguage) family	
Translated to Michelson	
Prototype	
DSL: Finance	
Process oriented, fully asynchronous	
Do not have a Smart Contract Language	https://github.com/LiskHQ

Sources for the languages are listed in the following document.

Milestone 1: Grobanalyse

rpfister102 edited this page a minute from now · 6 revisions

Analyse

Liste der bereits existierenden Smart Contract Programmiersprachen:

We selected the languages according to the following criteria.

- Wide-spread popular language (recommendations, blogs)
- Good documentation
- Popularity in GitHub (stars, watchers)

Bitcoin

- Ivy, https://github.com/ivy-lang/ivy-bitcoin - 255 Stars / 24 Watchers / 24 Forks
  - https://docs.ivy-lang.org/bitcoin/
  - Discussion Channel for any question regards to Ivy: https://discordapp.com/channels/396801556147732490/396801738813997056
- Simplicity, https://blockstream.com/simplicity.pdf
- Balzac, https://blockchain.unica.it/balzac/docs/

Ethereum

Pages 6

Find a Page...

Home

Meeting Minutes

Milestone 1: Grobanalyse

Milestone 2: Genaue Analyse

Milestone 3: Language Design

Milestone 4: ANTLR

+ Add a custom sidebar

Clone this wiki locally

https://github.com/bazo-b

- Solidity, https://solidity.readthedocs.io/en/develop/ - 5982 Stars / 504 Watchers / 1594 Forks
- Vyper, https://github.com/ethereum/vyper - 2007 Stars / 165 Watchers / 207 Forks
    - https://vyper.readthedocs.io/en/latest/installing-vyper.html
- Bamboo, https://github.com/pirapira/bamboo - 280 Stars / 35 Watchers / 35 Forks
- Flint, https://github.com/flintlang/flint - 170 Stars
    - https://docs.flintlang.org/
    - https://www.imperial.ac.uk/media/imperial-college/faculty-of-engineering/computing/public/ug-prizes-201718/Franklin-Schrans-A-new-programming-language-for-safer-smart-contracts.pdf
- Idris
    - https://publications.lib.chalmers.se/records/fulltext/234939/234939.pdf
    - https://www.idris-lang.org/documentation/
- L4, https://ethereumfoundation.org/devcon2/?session=designs-for-the-l4-contract-programming-language-based-on-deontic-modal-logic
- Babbage, https://medium.com/@chriseth/babbage-a-mechanical-smart-contract-language-5c8329ec5a0e
- Lolisa, https://arxiv.org/abs/1803.09885
- Serpent, https://www.cs.cmu.edu/~music/serpent/doc/serpent.htm
- eWASM, https://github.com/ewasm
- Mutan (deprecated, https://forum.ethereum.org/discussion/922/mutan

-faq)

- LLL, https://lll-docs.readthedocs.io/en/latest/lll_reference.html (supports core but hardly used)

Eternity

- Sophia, https://github.com/aeternity/protocol/blob/master/contracts/sophia.md
- Varna, https://cryptovarna.com

Other Blockchain

- Scilla (Zilliqa), https://github.com/Zilliqa/scilla - 58 Stars / 18 Watchers / 5 Forks
  - https://scilla-lang.org/
  - https://arxiv.org/pdf/1801.00687.pdf
- Michelson (Tezos), https://www.michelson-lang.com
- Liquidity (Tezos), http://www.liquidity-lang.org/doc/tutorial/tutorial.html - 89 Stars / 19 Watchers / 16 Forks
- Pact (Kadena), https://github.com/kadena-io/pact - 215 Stars / 22 Watchers / 24 Forks
  - https://github.com/kadena-io/pact
- Plutus, https://github.com/input-output-hk/plutus-prototype
- Marlowe, https://twitter.com/IOHK_Charles/status/963837766957137921
- RIDE, ttps://wavesplatform.com/files/docs/white_paper_waves_smart_contracts.pdf
- F*, https://www.fstar-lang.org
- Rholang, https://github.com/rchain/Rholang

    - https://developer.rchain.coop/tutorial/
- Lisk, https://lisk.io

Manche Blockchains nutzen auch normale Programmiersprachen: C, C++, C#, JS, Java, Kotlin, Rust, GoLang usw.

Quellen:

- https://github.com/s-tikhomirov/smart-contract-languages
- https://hackernoon.com/contractpedia-an-encyclopedia-of-40-smart-contract-platforms-4867f66da1e5
- https://blog.comae.io/smart-contract-languages-development-to-follow-992e30774b39
- https://hackernoon.com/comparison-of-smart-contract-platforms-2796e34673b7
- https://github.com/Overtorment/awesome-smart-contracts

Hilfreiche Links

- https://en.wikipedia.org/wiki/Programming_paradigm

Appendix B

Domain Analysis

DOMAIN ANALYSIS - STUDY WORK

SMART CONTRACTS IN BAZO BLOCKCHAIN

December 12, 2018

Keerthikan Thurairatnam & Remo Pfister
HSR - Hochschule für Technik Rapperswil
Department of Computer Science

Document History

Date	Vers.	Change(s)	Author
26.09.2018	1.0	Dokument erstellt	Remo
27.09.2018	1.1	Analysis Solidity	Remo
28.09.2018	1.2	Analysis Vyper	Keerthikan
04.10.2018	1.3	Analysis Scilla	Remo
07.10.2018	1.4	Analysis Scilla	Keerthikan
10.12.2018	1.5	Spell Check and Corrections	Remo

Contents

1 INTRODUCTION

1.1 Purpose

This document contains the domain analysis for our study. Three different smart contract languages are analyzed: Solidity, Vyper and Scilla.

1.2 Validity Period

The document is valid during the period of the "Studienarbeit HS 2018". Changes are recorded in the document history.

2 Blockchain Basics

A Blockchain basically consists of blocks of transactions which are chained one after the other. It is a distributed, transactional database. Everyone in the network can read entries from this database. Entries in the database are immutable. The database can only be extended by creating new transactions.

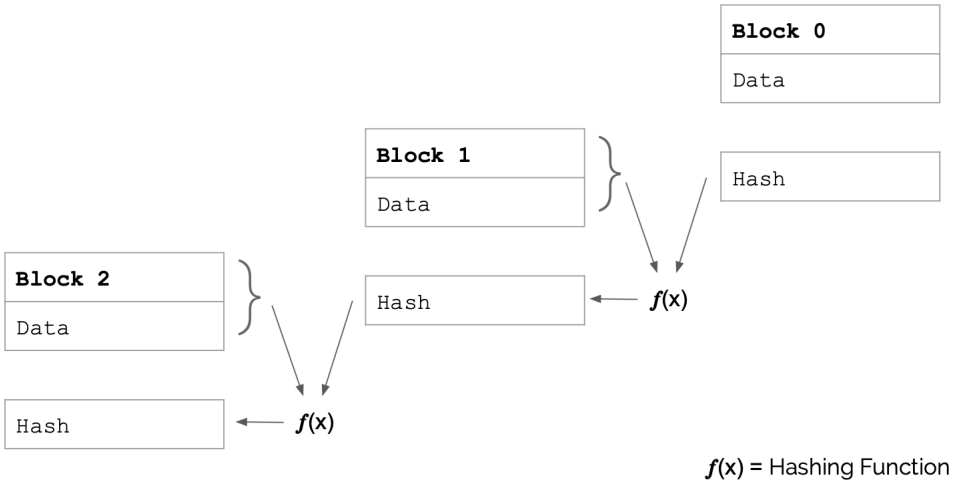

Figure 1: Blockchain

2.1 Transactions

As already mentioned, you need to create a **transaction**, which has to be accepted by all other participants of the network, to add new data in the database. The transaction is atomic, which means it is either processed completely or not at all. During the processing of a transaction, no other transaction can alter the database.

To verify the creator, each transaction is cryptographically signed, so authorization checks can be performed very easily.

2.2 Blocks

Transactions are bundled into a block. So a block is a collection of transactions and some meta information such as a timestamp, a nonce etc. New blocks will be mined and distributed

among all nodes in the network in a rather regular intervals - in Bitcoin, it is about 10 minutes, while in Ethereum it is between 10 to 19 seconds.

If two transactions contradict each other (e.g. double spending), the one that is processed first will become part of the block. The other one will be rejected.

2.3 Mining

Mining is the "order selection mechanism", which decides which block is added to the chain next. Due to this mechanism, it may happen that blocks are reverted from time to time to prevent branching. This only happens at the end of the chain. The more blocks that are added to the end, the less likely they are to be reverted.

2.4 Smart Contracts

Smart Contracts are self-executing contracts with the terms of the agreement between minimum two parties [2]. They are written in lines of code and are distributed among the decentralized blockchain network as transactions.

General requirements that have to be met for smart contracts:

- Receive and send coins
- Execute program logic when certain conditions are fulfilled
- Should be able to call other contracts (including themselves)
- Should be secure

3 SOLIDITY

3.1 Sources

This analysis is based on the solidity v0.4.25 documentation [6]. Note that we only picked the important parts from the documentation and left out details which are not important for our analysis in our opinion. More information can be found in the official documentation. Most of the examples used in this analysis are copied or rewritten from the examples given within the solidity documentation.

3.2 Restrictions

At the time of writing, the latest version of Solidity is v0.4.25. Therefore we focus on this version during our analysis. There is also a documentation for v0.5.0 available, but as this version is not released, we do not know if this documentation is complete or not, which can lead to misconception. We did not analyze this version.

3.3 Introduction & Background

"Solidity is a contract-oriented, high-level language for implementing smart contracts. It was influenced by C++, Python and JavaScript and is designed to target the Ethereum Virtual Machine (EVM).

Solidity is statically typed, supports inheritance, libraries and complex user-defined types among other features.

A contract in the sense of Solidity is a collection of code (its functions) and data (its state) that resides at a specific address on the Ethereum blockchain."[6]

3.4 Analysis

3.4.1 Versioning and Backward Compatibility

Solidity uses pragmas (instructions for the compiler) to ensure that a contract runs correctly and does not behave differently with different compiler versions.

3.4.2 Contracts

In Solidity, contracts are similar to classes in other languages such as Java or C#. Contracts can contain State Variables, Functions, Modifiers, Structs and Enums. Also inheritance is supported.

Inheritance The inheritance system is very similar to Python's, it also supports multi-inheritance, which not only brings benefits, but also comes with drawbacks such as the diamond problem.

As in other languages like C# that support inheritance, function calls are virtual, which means that (in general) the most derived function is called. This can be bypassed, by giving the contract name explicitly.

When inheritance is used, only a single contract is created on the blockchain. Code from the base contracts is copied into the new contract.

We will not go into any further details, so please consult the official documentation for more information about the diamond problem or the inheritance mechanism.

Abstract Contracts
Abstract contracts are used to make the interface of a contract known to the compiler.

```
1 contract Contact {
2     function getName() public returns (string name); // abstract function
3 }
```

Interfaces
Interfaces are also supported and have the same restrictions as in other languages:

- Can only contain function declarations, no implementations
- Cannot inherit
- Cannot define Array, Structs, Enums or Variables

```
interface Token {
    function transfer(address recipient, uint amount) public;
}
```

Contract implement interfaces the same way as they inherit other contracts, with the is keyword.

The new keyword
By using the new keyword with a contract, a contract can create a new contract during execution.

The this keyword
By using the this keyword the current contract can be accessed.

The selfdestruct(address recipient) function
This function can be used to destroy the current contract and send its funds to the given address.

Libraries
Libraries are contracts that are only deployed once at a specific address. Other contracts can call them using a special call feature provided by the EVM, which allows their code to be executed in the context of the calling contract. This way the library can access the calling contract and its storage. State variables can only be accessed if explicitly passed by the calling contract. Direct calls to library functions are only possible, if they do not modify the state. For more details, please consult the official solidity documentation.

Using For
Using A for B: attaches the library functions from A to any type B. The attached functions will receive the object they are called on as the first parameter. This is similar to the **self variable in Python**. There is also an option to attach library functions to any type in the contract using

a *-symbol instead of type B.

3.4.3 Types

Solidity is a statically and strong typed language. Supported types are shown below.

Table 1: Value Types

Type	Keyword
Boolean	bool
Signed Integer	int[1]
Unsigned Integer	uint[1]
Fixed Point Numbers	fixedMxN[2]
Unsigned Fixed Point Numbers	ufixedMxN[2]
Address (160 Bit)	address
Fixed-Size byte array	bytes[3]
Dynamically-Size byte array	bytes or string
Enum	enum

Table 2: Reference Types

Type	Remarks
Arrays	Array of any type
Structs	Define new types
Mappings	cf. hash tables

[1]: Default length is 256 bit. Integer size can be specified by using int8 to int256 for signed integers or uint8 to uint256 for unsigned integers. Both types can be increased in steps of 8.
[2]: M stands for the number of bits taken by the type and N for how many decimal points are available. M can be any value from 8 to 256 in steps of 8. `fixed and ufixed` are aliases for `fixed128x18 and ufixed128x18`. Fixed Point Numbers are not fully supported by Solidity. They can be declared but you can not assign to or from them.
[3]: Options are `bytes1, bytes2, bytes3, ..., bytes32 and bytes`. `bytes` is an alias for `bytes1`.

Fixed Point Numbers vs Floating Points:
"The main difference between floating point (...) and fixed point numbers is that the number of bits used for the integer and the fractional part (the part after the decimal dot) is flexible in the former, while it is strictly defined in the latter. Generally, in floating point almost the entire space is used to represent the number, while only a small number of bits define where the decimal point is."[6]

Addresses
Addresses are used to store the address of contracts or key value pairs (e.g. Wallets). They

disallow arithmetic operations. From version 0.5.0 contracts do not derive from the address type any longer, but they can still be converted. An address has several members such as balance, to get the balance of an address, or transfer to transfer Ether. There are further members, for more details, please refer to the solidity v0.4.25 documentation.

Mappings

Mappings are virtually initialized with every possible key (value equals zero). When querying a map for a key which is not inside the map, Solidity returns the zero value and does not throw an exception. This means loops over all values of a mapping are not possible and keys need to be remembered somehow.

3.4.4 Data location:

There are three different spaces, where data can be stored: **Storage, Memory and CallData** (Non-Modifiable and Non-Persistent)

The data location changes how assignments in smart contracts behave.

```
    pragma solidity ^0.4.0;

contract C {
    uint[] x; // the data location of x is storage

    // the data location of memoryArray is memory
    function f(uint[] memoryArray) public {
        x = memoryArray; // works, copies the whole array to storage
        var y = x; // works, assigns a pointer, data location of y is storage
        y[7]; // fine, returns the 8th element
        y.length = 2; // fine, modifies x through y
        delete x; // fine, clears the array, also modifies y
        // The following does not work; it would need to create a new temporary /
        // unnamed array in storage, but storage is "statically" allocated:
        // y = memoryArray;
        // This does not work either, since it would "reset" the pointer, but there
        // is no sensible location it could point to.
        // delete y;
        g(x); // calls g, handing over a reference to x
        h(x); // calls h and creates an independent, temporary copy in memory
    }
```

```
22
23      function g(uint[] storage storageArray) internal {}
24      function h(uint[] memoryArray) public {}
25  }
```

3.4.5 Variables

Variables are initialized with a default value when being declared without an explicit value assignment. The **default value represents the zero state of the type** (e.g. false for boolean, 0 for integers).

Mutable and Immutable Variables

Solidity provides both, mutable and immutable (constant) state variables. To define a immutable state variable, use the constant keyword. Constants have to be assigned from an expression which can be evaluated at compile time.

Variables in Functions

Variables declared in functions belong to the functions scope. The following code snippet is invalid code:

```
1   pragma solidity ^0.4.16;
2
3   contract ScopingErrors {
4       function scoping() public {
5           uint i = 0;
6
7           while (i++ < 1) {
8               uint same1 = 0;
9           }
10
11          while (i++ < 2) {
12              uint same1 = 0;// Illegal, second declaration of same1
13          }
14      }
```

Additionally, variables are initialized at the beginning of a function to their default value, so code like this is legal although it is not very readable:

```
pragma solidity ^0.4.0;

contract C {
    function foo() public pure returns (uint) {
        // baz is implicitly initialized as 0
        uint bar = 5;
        if (true) {
            bar += baz;
        } else {
            uint baz = 10;// never executes
        }
        return bar;// returns 5
    }
}
```

Solidity will change this in version 0.5.0 and use block scoping instead.

Global Variables

Global variables are used to access the blockchain.

msg The message which called the contract.

tx The current transaction.

block The current block.

*Note: "The values of all members of msg, including msg.sender and msg.value, can change for every **external** function call. This includes calls to library functions."*[6]

Units

Solidity supports two types of units: **Ether Units and Time Units**. The first are used to convert between ether, wei, finney and szabo. The latter are used to work with time. Seconds, minutes, hours, days, weeks and years are supported.

ABI Encoding Functions

Contracts can use ABI encoding functions which are used to encode given arguments for the

ABI. Those functions are used for function calls without actually calling the function directly. For further detail, please consult the official solidity documentation.

3.4.6 Functions

Visibility

public Part of the Contract Interface. Can be called internally and externally.

private Not part of the Contract Interface. Are only visible in the contract they are defined in, not in derived contracts. Can only be called internally.

external Part of the Contract Interface. Can only be called externally.

interal Not Part of the Contract Interface. Are visible to contract and derived contracts. Can only be called internally.

Scoping
Variables declared in functions belong to the functions scope as already mentioned. So declaring a variable twice within a function body is not possible, even if it is done in separate blocks. This will change in v0.5.0 of Solidity.

Constructor
The constructor is **only called once during the creation of the contract**. It cannot be called afterwards.
Only one constructor is allowed, i.e. constructor overloading is not supported. If no constructor is defined, the default constructor will automatically be generated.

Getter Functions
Getter functions are automatically generated for public state variables. They have external visibility. If variables are called internally, the state variable are accessed directly.

Returning
Functions in Solidity can return multiple values. The return types have to be declared within the function header.

Overloading

Function overloading is supported.

Overriding

Due to inheritance, overriding is supported. To overwrite a function of the base class, a function with the same name and number/types of parameters need to be defined. The output parameters must be the same, otherwise an error is caused.

Lambdas

Lambdas are currently not supported.

Views, Pure

Functions declared *view* promise that they do not change the state of the contract (Provable). Functions declared *pure* promise not to access (read or change) the state.

Fallback function

There can be one unnamed function without arguments or return value in a contract which serves as a fallback function. This function is called if no other function in the contract matches the given function identifier.

Function Types

```
1    function (<parameter types>)
2             {internal|external}
3             [pure|constant|view|payable]
4             [returns (<return types>)]
```

Function Modifiers

Modifiers can be used to extend the behaviour of functions. Often, they are used for checking preconditions and postconditions of a function. A modifier is defined using the modifier keyword within a contract and can be applied to a function within the function header. Multiple modifiers can be applied at the same time. When the _-symbol in a modifier is reached, the actual function is executed. After the return statement in the function, the execution jumps back into the modifier after the _-symbol. Modifiers are inheritable properties.

```
pragma solidity ^0.4.22;

contract BankAccount {
    function withdraw() public isOwner {...}

    modifier isOwner {
        require(
            msg.sender == owner,
            "You must be the owner to call this function."
        );
        _;
    }
}
```

Payable
The payable keyword marks a function to allow to receive Ether with a call. Otherwise the Ethers are rejected.

3.4.7 Events

Events are declared within the contract. They specify an event name and its parameters. An Event is triggered within a function by using the emit keyword. User interfaces and server applications can listen to such an event and register handlers for them. Other contracts cannot. As soon as the event is emitted, the watchers are triggered and receive the arguments specified in the event.

3.4.8 Type Deduction

It is not required to explicitly specify the type of a variable. The compiler can infer it from the type of the first expression assigned to this variable.

Implicit Conversion
Is executed if an operator is applied to different types. The compiler tries to convert one of the operands to the type of the other.

Explicit Conversion

Explicit conversion can be done by wrapping the assigned value with the explicit type. This can lead to unexpected behaviour. If a type is casted to a smaller type the higher-order bits are cut off.

3.4.9 Error Handling

There are two types of exceptions: assert-style exceptions and require-style exceptions. Internally, revert operations are performed in both cases. There is no way to continue the execution safely. Code execution can be aborted and state can be reverted by explicitly calling the revert function.

3.4.10 Integrated functions

Solidity supports several mathematical and cryptographic functions such as sha256, ripemd160 and addmod.

3.4.11 Inline Assembly

Solidity supports inline assembly in order to support legacy code.

3.5 Limitations

- Keywords are restricted to ASCII character set.
- String values can contain UTF-8 encoded data.
- Functions and state variables are in the same namespace.
- Max. recursion depth: 1024

4 ANALYZE VYPER

4.1 Sources

This analysis is based on the Vyper v0.1.0-beta3 documentation [7]. Note that we only picked the important parts from the documentation and left out details which are not important for our analysis in our opinion. More information can be found in the official documentation. Most of the examples used in this analysis are copied or rewritten from the examples given within the vyper documentation.

4.2 Restrictions

At the time of writing, Vyper is in beta development (v0.1.0-beta3)[7]. Therefore, the language specification may vary in the final release.

4.3 Introduction & Background

Vyper is a contract-oriented and python-like programming language for the Ethereum Virtual Machine (EVM). The project was started in late 2016 and is still under development.

The main principles and goals of creating a new language are the following:

- **Security**: Provide built-in checks to create more secure "smart" contracts.
- **Language and compiler simplicity**: Remove unnecessary features and keep the language simple.
- **Auditability**: The language should be human-readable and avoid misconception.

4.4 Analysis

4.4.1 Types

Vyper is a statically and strong typed language. Supported types are shown below.

Table 3: Value Types

Type	Keyword
Boolean	bool
Signed Integer	int128
Unsigned Integer	uint256
Decimal	decimal
Address (160bit)	address
Units	units{}
32-bit-wide Byte Array	bytes32
Fixed-size Byte Array	bytes[Length]

Table 4: Reference Types

Type	Remarks
Fixed-size Lists	Array of any type
Structs	Group several variables
Mapping	cf. hash tables

4.4.2 Visibilities

Vyper supports only two visibilities (aka. access control modifiers), namely **public** and **private**.

4.4.3 Variables

A variables is declared with an identifier, a data type and optionally a visibility (default: private). Note that the **public** visibility means that the variable is *readable* by an external caller, but not *writable*.

```
1    value: public(wei_value)
2    seller: public(address)
3    total_paid: int128
```

4.4.4 Built-in Global Variables

block provides information about the block at the time of calling

msg provides information on the *method caller*

Warning: If a method is called from outside, the **msg.sender** is set to the actual caller for the first time. However, if that method calls another method within the same contract, **msg.sender** will be set to the contract itself.

4.4.5 Functions

The syntax of a function is similar to Python. Apart from that, the functions in Vyper must be annotated with a visibility, either **@public** or **@private**.

```
1    @public
2    @payable
3    def bid():
4        // ...
```

By using the **@payable** annotation, it indicates that the *bid*-function is only executed when the message calling the contract is sent with Ether. Furthermore, **@constant** decorator can also be used to declare that the method only reads the contract state or return a simple calculation without changing the state. Note that reading the blockchain state is free, modifying costs gas. Thus, adding **@constant** annotation provides additional certainty of saving gas fees.

Constructor & Destructor

There are two special type of functions.

__init__() Constructor initializes a new contract for use. Arguments can be defined, if needed.

selfdestruct(address) Refunds the receiver at the defined address and destroys the contract.

Default function

When a contract is called with an undefined function identifier, the default function will be executed. Default function is the same as Solidity's fallback function 3.4.6.

```
1    @public
2    @payable
3    def __default__():
4        // ...
```

The default function's identifier should be always named "*__default__*" and annotated with *@public*. It cannot have parameters and cannot return anything. Additionally, if the function is annotated as *@payable*, it will be executed whenever the contract is sent Ether. It is

important to mention that Ethereum does not differentiate between sending Ether to a user's address or to a contract.

If no default function is defined, Vyper generates a default function, just as in Solidity, and calls **REVERT** opcode. It produces an exception, so the funds will not be transferred to the receiver.

Best Practice - Structure of Functions

It is recommended to structure the function into three phases:

1. Checking conditions
2. Performing actions (potentially changing conditions)
3. Interacting with other contracts

An example code snippet is shown below.

```
    @public
    @payable
    def end_auction():
        # 1. Conditions
        assert block.timestamp >= self.auction_end
        assert not self.ended

        # 2. Effects - change state variable
        self.ended = True

        # 3. Interaction
        send(self.beneficiary, self.highest_bid)
```

4.4.6 Events

Events are used to notify the subscribers when something of interest occurs. Events must be declared before global declarations and function definitions as shown below.

```
    Payment: event({amount: int128, from: indexed(address)})

    total_paid: int128

```

```
    @public
    @payable
    def pay():
        self.total_paid += msg.value
        log.Payment(msg.value, msg.sender) # emit event
```

Events do not take storage, so they do not cost gas either. The drawback is that contracts cannot listen to the events. They are available only to clients.

4.4.7 Control Structures

Vyper supports only the following control structures.

- If-Else
- for-Loop
- continue, break
- return

```
    for i range(start, end, incr):
        if i >= self.nextFunderIndex:
            // ...
        else:
            // ...
```

Vyper does not support *while*-loop. Looping over a variable may cause infinite-loop and make *gas limit attacks* possible. Therefore, Vyper supports only a simple ranged *for*-loop. As a consequence, Vyper is not a turing-complete language.

4.4.8 Special features

Bounds checks Check if the array access lies within the range

Under/overflow checks Check the over and underflow on the arithmetic operations

4.4.9 Unsupported Features - Address the Problems with Solidity

The following features are not supported to avoid misleading or difficult to understand code.

Function modifiers In Solidity, function modifiers can be used to check preconditions and postconditions, as shown below. They can be misleading and harm auditability. In Vyper, it is recommended to use inline assert checks.

```
modifier onlyOwner {
    require(msg.sender == owner, "Only owner can call this function.");
    _; // Mandatory!! The function body is inserted here.
}

function close() public onlyOwner {
    selfdestruct(owner);
}
```

Class Inheritance Some implementation is "hidden" in the superclass. It can negatively impact auditability.

Inline assembly Solidity lets you write "inline assembly" inside Solidity source code. Assembly codes are hard to read and debug. Refactoring the code later is tedious.

```
function at(address _addr) public view returns (bytes o_code) {
    assembly {
        let size := extcodesize(_addr)
        o_code := mload(0x40)
        mstore(0x40, add(o_code, and(add(add(size, 0x20), 0x1f), not(0x1f))))
        mstore(o_code, size)
        extcodecopy(_addr, add(o_code, 0x20), 0, size)
    }
}
```

Function overloading It is easier to write misleading code. One function *log("hello")* just logs a message but the other function *log("hello", "world")* could execute harmful operations (e.g. steal money).

Operator overloading enables redefining popular operators (e.g. +, - etc.) and give rise to misconception. For instance, one could override the "+" arithmetic operator and

execute harmful operations behind the scene.

Recursive calling makes it impossible to set an upper bound on gas limits which may lead to *gas limit attacks*.

Infinite-length loops Similar to recursive calling. Therefore, Vyper supports only limited *for*-loop, the range of which is pre-determined.

Binary fixed point Decimal fixed point is better because it has an exact representation. On the other hand, binary fixed point often requires approximation:

$$(0.2)_{10} = (0.001100110011...)_2$$

Approximation could lead to unexpected results, e.g.

$$0.3 + 0.3 + 0.3 + 0.1 \neq 1$$

.

5 SCILLA

5.1 Sources

This analysis is based on the Scilla v0.0.1 documentation [4] and the corresponding whitepaper [1]. Note that we only picked the, from our point of view, important parts and left out irrelevant details. More information about Scilla can be found in the official documentation or the whitepaper. Most of the code examples are copies or rewrites of the scilla documentation.[5]

5.2 Restrictions

This analysis focuses on the version 0.0.1 of scilla-lang. Knowledge of Haskell, Functional Languages and the lambda calculus are required as we will not explain any of these languages or concepts in this analysis.

5.3 Introduction & Background

"Scilla, short for Smart Contract Intermediate-Level Language, is an intermediate-level smart contract language being developed for Zilliqa. Scilla has been designed as a principled language with smart contract safety in mind.

Scilla imposes a structure on smart contracts that will make applications less vulnerable to attacks by eliminating certain known vulnerabilities directly at the language-level. Furthermore, the principled structure of Scilla will make applications inherently more secure and amenable to formal verification.

The language is being developed hand-in-hand with formalization of its semantics and its embedding into the Coq proof assistant — a state-of-the art tool for mechanized proofs about properties of programs."[5]

5.4 Analysis

5.4.1 Design Principles

Seperation between Computation and Communication
"Contracts in Scilla are structured as communicating automata: every in-contract computation (e.g., changing its balance or computing a value of a function) is implemented as a standalone, atomic transition, i.e., without involving any other parties. Whenever such involvement is required (e.g., for transferring control to another party), a transition would end, with an explicit communication, by means of sending and receiving messages. The automata-based structure makes it possible to disentangle the contract-specific effects (i.e., transitions) from blockchain-wide interactions (i.e., sending/receiving funds and messages), thus providing a clean reasoning mechanism about contract composition and invariants."[4]
Seperation between Effectful and Pure Computations
"Any in-contract computation happening within a transition has to terminate, and have a predictable effect on the state of the contract and the execution. In order to achieve this, Scilla draws inspiration from functional programming with effects, drawing a distinction between pure expressions (e.g., expressions with primitive data types and maps), impure local state manipulations (i.e., reading/writing into contract fields) and blockchain reflection (e.g., reading current block number). By carefully designing semantics of interaction between pure and impure language aspects, Scilla ensures a number of foundational properties about contract transitions, such as progress and type preservation, while also making them amenable to interactive and/or automatic verification with standalone tools."[4]
Seperation between Invocation and Continuation
"Structuring contracts as communicating automata provides a computational model, which only allows tail-calls, i.e., every call to an external function (i.e., another contract) has to be done as the absolutely last instruction."[4]

5.4.2 Types

Lists
There are two structural recursion primitives as known from other languages like Haskell: *list_foldl* and *list_foldr*.

Table 5: Primitive Data Types

Type	Keyword
Signed Integer	Int32, Int64, Int128
Unsigned Integer	Uint32, Uint64, Uint128
Strings	String
Hashes	ByStr32
Address (160bit)	ByStr20
Block Numbers	BNum
Maps	Map

Table 6: Algebraic Data Types

Type	Keyword	Remarks
Boolean	Bool	Value is True or False
Option	Some	Checks for Presence
Option	None	Checks for Absence
List	Nil	Creates empty list
List	Cons	Adds elements to existing list
Pair	Pair	Creates a pair
Nat	Zero or Succ Nat	Work with natural numbers

5.4.3 Standard Libraries

Scilla comes with four standard libraries which can be used to write smart contracts, such as *BoolUtils, ListUtils, NatUtil* and *PairUtils*.

5.4.4 State Variables

Scilla supports mutable and immutable state variables. Both are initialized in the construction phase. The difference is, that mutable state variables can be modified by transitions/continuations, while immutables cannot be modified at all.

```
(* Immutable fields declaration *)

(vname_1 : vtype_1,
 vname_2 : vtype_2)

(* Mutable fields declaration *)

```

```
8 field vname_1 : vtype_1 = init_val_1
9 field vname_2 : vtype_2 = init_val_2
```

5.4.5 Expressions

Expressions handle pure operations. There are 9 different types of expressions:

- Global binding of a variable to another expression
- Local binding of a variable to another expression
- Messages
- Functions
- Type Functions
- Variable Instantiation
- Application
- Application of Built-in functions
- Patterns (Match Expressions)

5.4.6 Statements

Statements are operations which access or modify the state (impure).

Reading from Blockchain State
In Scilla, you can read from the blockchain state. This state consists of blocks (keys) associated with their values such as the block number.

Accepting/Rejecting incoming payments
Incoming payments have to be accepted by using the *accept* statement. Without invoking this statement, the transition does not accept the payment and rejects it.

5.4.7 Transitions / Functions

Transitions in Scilla are similar to functions/methods in other languages. They are used to change the state of a contract.

```
// Mutable state
field welcome_msg : String = ""

transition setHello (msg : String)
  welcome_msg := msg;
end
```

5.4.8 Communication

Contracts can communicate with each other through the **send** statement. Such calls to external contracts must be done as absolutely last instruction, as shown below.

```
transition getHello ()
    r <- welcome_msg;
    msg = {_tag : Main; _recipient : _sender; _amount : 0; msg : r};
    msgs = one_msg msg;
    send msgs
end
```

5.4.9 Continuation

It is quite common that a contract calls a function from another contract for computation. Once completed, it will use the result of the call in the rest of the execution. However, Scillas' contract model prevent this as there are no instructions allowed after the **send** statement at the end a transition. To solve this problem, Scilla uses an approach called **Continuation-passing style (CPS)**, as shown below.

```
(* Specifying a continuation in a Caller contract *)
continuation UseResult (res: uint)
    send(<to -> owner, amount -> 0, tag -> "main", msg ->res>, MT)

(* Using a continuation in a transition of Caller *)
transition ClientTransition
    (sender : address, value : uint, tag : string)
    (* code of the transition *)
    send (<to -> sender, amount -> 0, tag → "main", msg -> res>, UseResult)
    (* Returning a result in a callee contract *)
```

```
end

transition ServerTransition
    (sender : address, value : uint, tag : string)
    return value
end
```

The main difference between continuation and transition is that continuations are "passive", i.e. they are invoked only after the result is returned from the callee's contract. On the other hand, transitions are "active" and could be invoked by sending a message.

5.4.10 Events

The contract can also communicate with the clients (off-chain) using the **event** statement.

```
e = { _eventname : "eventName"; <entry>_2 ; <entry>_3 };
event e;
```

5.4.11 Advantages & Disadvantages

Advantages

- Provides clear separation between the communication aspect and its programming concept
- Prevents Re-Entrancy Attack by using CPS pattern (continuation-passing style)
- Provides formal verification tool to prove safety and liveness properties of the contract
- Since it is an intermediate language, high-level languages (e.g. Solidity) can also be translated to Scilla to perform program analysis and verification

6 Security Issues/Consideration

Most of the problems described in this section only apply to Solidity, as the other two languages already solve those problems. Nevertheless those issues and attacks have to be considered when writing a new language.

6.1 Re-entrancy attack

Note: The code examples here are taken from the Zilliqa Blog post "Scilla Design Story Piece by Piece: Part 1" by Amrit Kumar. [3]

```
contract UnsafeContract1{
    mapping(address => uint) shares;

    function withdraw() public {
        if (msg.sender.call.value(shares[msg.sender])())
            shares[msg.sender] = 0; // update state after the external invokation
    }
}
```

"In the *UnsafeContract1* example, the contract sends out a message to transfer the share to the sender (via *msg.sender.call.value()*) and then sets the share to 0 by updating shares in the next line.

If the callee address is a contract, it can invoke *withdraw()* method back again. Notice that in *withdraw()*, the caller's entry in shares is updated to 0, only after the external call has terminated. When the malicious contract calls back *withdraw()*, the shares of the sender will not be updated. This allows the malicious contract to withdraw its share multiple times until the provided gas is consumed.

If the recipient of the message had been a user (not a contract), then it would not have been possible to call back into the contract and hence the execution would have ended as expected."[3]

A possible fix for the re-entrance-attack is shown below.

```
contract FixedContract1{
    // Mapping of address and amount
    mapping(address => uint) shares;
    // Withdraw a share
    function withdraw() public {
       uint share = shares[msg.sender];
       shares[msg.sender] = 0;
       msg.sender.transfer(share);
    }
}
```

6.2 Gas Limit and Loops

In the following example, the *while*-loop does not have a fix number of iterations. If they run too long, it might require more gas than available in the block. In this case, the whole block execution will always fail and the contract gets stuck completely.

```
while (voters[to].delegate != address(0)) {
    to = voters[to].delegate;

    // We found a loop in the delegation, not allowed.
    require(to != msg.sender, "Found loop in delegation.");
}
```

See 7.1.1 Voting example for the full source code.

6.3 Integer overflow and underflow

Once the maximum of *uint* type is reached, it starts back from 0. Due to this, a recipient can lose a considerable amount of money. It is recommended to do overflow and underflow checks before doing any arithmetic operations.

```
mapping (address => uint256) public balanceOf;

// INSECURE
function transfer(address _to, uint256 _value) {
```

```
    /* Check if sender has balance */
    require(balanceOf[msg.sender] >= _value);

    balanceOf[msg.sender] -= _value;
    balanceOf[_to] += _value; // balance of recipient is not checked
}
```

6.4 Miscellaneous

- tx.origin should not be used for authorization, use msg.sender instead.
- Callstack Depth (Fixed: all gas would be consumed well before reaching the 1024 call depth limit).
- Sending and receiving Ether using *send(), transfer(), and call.value()()*.

7 EXAMPLES

7.1 Solidity Examples

All the examples below are taken from the official Solidity documentation.[6]

7.1.1 Voting

The following example shows a **Ballot** voting contract[6]. The creator of the contract gives the right to vote each address individually. The voters can either vote themselves or another person they trust. At the end of the voting time, the winning proposal will be returned.

```
pragma solidity >=0.4.22 <0.6.0;

/// @title Voting with delegation.
contract Ballot {
    // This declares a new complex type which will
    // be used for variables later.
    // It will represent a single voter.
    struct Voter {
        uint weight; // weight is accumulated by delegation
        bool voted;  // if true, that person already voted
        address delegate; // person delegated to
        uint vote;   // index of the voted proposal
    }

    // This is a type for a single proposal.
    struct Proposal {
        bytes32 name;   // short name (up to 32 bytes)
        uint voteCount; // number of accumulated votes
    }

    address public chairperson;

    // This declares a state variable that
    // stores a `Voter` struct for each possible address.
    mapping(address => Voter) public voters;

    // A dynamically-sized array of `Proposal` structs.
```

```
28      Proposal[] public proposals;
29
30      /// Create a new ballot to choose one of `proposalNames`.
31      constructor(bytes32[] memory proposalNames) public {
32          chairperson = msg.sender;
33          voters[chairperson].weight = 1;
34
35          // For each of the provided proposal names,
36          // create a new proposal object and add it
37          // to the end of the array.
38          for (uint i = 0; i < proposalNames.length; i++) {
39              // `Proposal({...})` creates a temporary
40              // Proposal object and `proposals.push(...)`
41              // appends it to the end of `proposals`.
42              proposals.push(Proposal({
43                  name: proposalNames[i],
44                  voteCount: 0
45              }));
46          }
47      }
48
49      // Give `voter` the right to vote on this ballot.
50      // May only be called by `chairperson`.
51      function giveRightToVote(address voter) public {
52          // If the first argument of `require` evaluates
53          // to `false`, execution terminates and all
54          // changes to the state and to Ether balances
55          // are reverted.
56          // This used to consume all gas in old EVM versions, but
57          // not anymore.
58          // It is often a good idea to use `require` to check if
59          // functions are called correctly.
60          // As a second argument, you can also provide an
61          // explanation about what went wrong.
62          require(
63              msg.sender == chairperson,
64              "Only chairperson can give right to vote."
65          );
66          require(
67              !voters[voter].voted,
68              "The voter already voted."
69          );
```

```
70          require(voters[voter].weight == 0);
71          voters[voter].weight = 1;
72      }
73
74      /// Delegate your vote to the voter `to`.
75      function delegate(address to) public {
76          // assigns reference
77          Voter storage sender = voters[msg.sender];
78          require(!sender.voted, "You already voted.");
79
80          require(to != msg.sender, "Self-delegation is disallowed.");
81
82          // Forward the delegation as long as
83          // `to` also delegated.
84          // In general, such loops are very dangerous,
85          // because if they run too long, they might
86          // need more gas than is available in a block.
87          // In this case, the delegation will not be executed,
88          // but in other situations, such loops might
89          // cause a contract to get "stuck" completely.
90          while (voters[to].delegate != address(0)) {
91              to = voters[to].delegate;
92
93              // We found a loop in the delegation, not allowed.
94              require(to != msg.sender, "Found loop in delegation.");
95          }
96
97          // Since `sender` is a reference, this
98          // modifies `voters[msg.sender].voted`
99          sender.voted = true;
100         sender.delegate = to;
101         Voter storage delegate_ = voters[to];
102         if (delegate_.voted) {
103             // If the delegate already voted,
104             // directly add to the number of votes
105             proposals[delegate_.vote].voteCount += sender.weight;
106         } else {
107             // If the delegate did not vote yet,
108             // add to her weight.
109             delegate_.weight += sender.weight;
110         }
111     }
```

```
112
113     /// Give your vote (including votes delegated to you)
114     /// to proposal `proposals[proposal].name`.
115     function vote(uint proposal) public {
116         Voter storage sender = voters[msg.sender];
117         require(!sender.voted, "Already voted.");
118         sender.voted = true;
119         sender.vote = proposal;
120
121         // If `proposal` is out of the range of the array,
122         // this will throw automatically and revert all
123         // changes.
124         proposals[proposal].voteCount += sender.weight;
125     }
126
127     /// @dev Computes the winning proposal taking all
128     /// previous votes into account.
129     function winningProposal() public view
130             returns (uint winningProposal_)
131     {
132         uint winningVoteCount = 0;
133         for (uint p = 0; p < proposals.length; p++) {
134             if (proposals[p].voteCount > winningVoteCount) {
135                 winningVoteCount = proposals[p].voteCount;
136                 winningProposal_ = p;
137             }
138         }
139     }
140
141     // Calls winningProposal() function to get the index
142     // of the winner contained in the proposals array and then
143     // returns the name of the winner
144     function winnerName() public view
145             returns (bytes32 winnerName_)
146     {
147         winnerName_ = proposals[winningProposal()].name;
148     }
149 }
```

7.1.2 Simple Open Auction

```
pragma solidity >=0.4.22 <0.6.0;

contract SimpleAuction {
    // Parameters of the auction. Times are either
    // absolute unix timestamps (seconds since 1970-01-01)
    // or time periods in seconds.
    address payable public beneficiary;
    uint public auctionEndTime;

    // Current state of the auction.
    address public highestBidder;
    uint public highestBid;

    // Allowed withdrawals of previous bids
    mapping(address => uint) pendingReturns;

    // Set to true at the end, disallows any change.
    // By default initialized to `false`.
    bool ended;

    // Events that will be emitted on changes.
    event HighestBidIncreased(address bidder, uint amount);
    event AuctionEnded(address winner, uint amount);

    // The following is a so-called natspec comment,
    // recognizable by the three slashes.
    // It will be shown when the user is asked to
    // confirm a transaction.

    /// Create a simple auction with `_biddingTime`
    /// seconds bidding time on behalf of the
    /// beneficiary address `_beneficiary`.
    constructor(
        uint _biddingTime,
        address payable _beneficiary
    ) public {
        beneficiary = _beneficiary;
        auctionEndTime = now + _biddingTime;
    }
```

```
40
41      /// Bid on the auction with the value sent together with this transaction.
42      /// The value will only be refunded if the auction is not won.
43      function bid() public payable {
44          // No arguments are necessary, all
45          // information is already part of
46          // the transaction. The keyword payable
47          // is required for the function to
48          // be able to receive Ether.
49
50          // Revert the call if the bidding
51          // period is over.
52          require(
53              now <= auctionEndTime,
54              "Auction already ended."
55          );
56
57          // If the bid is not higher, send the money back.
58          require(
59              msg.value > highestBid,
60              "There already is a higher bid."
61          );
62
63          if (highestBid != 0) {
64              // Sending back the money by simply using
65              // highestBidder.send(highestBid) is a security risk
66              // because it could execute an untrusted contract.
67              // It is always safer to let the recipients
68              // withdraw their money themselves.
69              pendingReturns[highestBidder] += highestBid;
70          }
71          highestBidder = msg.sender;
72          highestBid = msg.value;
73          emit HighestBidIncreased(msg.sender, msg.value);
74      }
75
76      /// Withdraw a bid that was overbid.
77      function withdraw() public returns (bool) {
78          uint amount = pendingReturns[msg.sender];
79          if (amount > 0) {
80              // It is important to set this to zero because the recipient
81              // can call this function again as part of the receiving call
```

```
            // before `send` returns.
            pendingReturns[msg.sender] = 0;

            if (!msg.sender.send(amount)) {
                // No need to call throw here, just reset the amount owing
                pendingReturns[msg.sender] = amount;
                return false;
            }
        }
        return true;
    }

    /// End the auction and send the highest bid
    /// to the beneficiary.
    function auctionEnd() public {
        // It is a good guideline to structure functions that interact
        // with other contracts (i.e. they call functions or send Ether)
        // into three phases:
        // 1. checking conditions
        // 2. performing actions (potentially changing conditions)
        // 3. interacting with other contracts
        // If these phases are mixed up, the other contract could call
        // back into the current contract and modify the state or cause
        // effects (ether payout) to be performed multiple times.
        // If functions called internally include interaction with external
        // contracts, they also have to be considered interaction with
        // external contracts.

        // 1. Conditions
        require(now >= auctionEndTime, "Auction not yet ended.");
        require(!ended, "auctionEnd has already been called.");

        // 2. Effects
        ended = true;
        emit AuctionEnded(highestBidder, highestBid);

        // 3. Interaction
        beneficiary.transfer(highestBid);
    }
}
```

7.1.3 Blind Auction

```
pragma solidity >0.4.23 <0.5.0;

contract BlindAuction {
    struct Bid {
        bytes32 blindedBid;
        uint deposit;
    }

    address payable public beneficiary;
    uint public biddingEnd;
    uint public revealEnd;
    bool public ended;

    mapping(address => Bid[]) public bids;

    address public highestBidder;
    uint public highestBid;

    // Allowed withdrawals of previous bids
    mapping(address => uint) pendingReturns;

    event AuctionEnded(address winner, uint highestBid);

    /// Modifiers are a convenient way to validate inputs to
    /// functions. `onlyBefore` is applied to `bid` below:
    /// The new function body is the modifier's body where
    /// `_` is replaced by the old function body.
    modifier onlyBefore(uint _time) { require(now < _time); _; }
    modifier onlyAfter(uint _time) { require(now > _time); _; }

    constructor(
        uint _biddingTime,
        uint _revealTime,
        address payable _beneficiary
    ) public {
        beneficiary = _beneficiary;
        biddingEnd = now + _biddingTime;
        revealEnd = biddingEnd + _revealTime;
    }
```

```
40
41      /// Place a blinded bid with `_blindedBid` =
42      /// keccak256(abi.encodePacked(value, fake, secret)).
43      /// The sent ether is only refunded if the bid is correctly
44      /// revealed in the revealing phase. The bid is valid if the
45      /// ether sent together with the bid is at least "value" and
46      /// "fake" is not true. Setting "fake" to true and sending
47      /// not the exact amount are ways to hide the real bid but
48      /// still make the required deposit. The same address can
49      /// place multiple bids.
50      function bid(bytes32 _blindedBid)
51          public
52          payable
53          onlyBefore(biddingEnd)
54      {
55          bids[msg.sender].push(Bid({
56              blindedBid: _blindedBid,
57              deposit: msg.value
58          }));
59      }
60
61      /// Reveal your blinded bids. You will get a refund for all
62      /// correctly blinded invalid bids and for all bids except for
63      /// the totally highest.
64      function reveal(
65          uint[] memory _values,
66          bool[] memory _fake,
67          bytes32[] memory _secret
68      )
69          public
70          onlyAfter(biddingEnd)
71          onlyBefore(revealEnd)
72      {
73          uint length = bids[msg.sender].length;
74          require(_values.length == length);
75          require(_fake.length == length);
76          require(_secret.length == length);
77
78          uint refund;
79          for (uint i = 0; i < length; i++) {
80              Bid storage bidToCheck = bids[msg.sender][i];
81              (uint value, bool fake, bytes32 secret) =
```

```
                    (_values[i], _fake[i], _secret[i]);
            if (bidToCheck.blindedBid != keccak256(abi.encodePacked(value, fake, secret))) {
                // Bid was not actually revealed.
                // Do not refund deposit.
                continue;
            }
            refund += bidToCheck.deposit;
            if (!fake && bidToCheck.deposit >= value) {
                if (placeBid(msg.sender, value))
                    refund -= value;
            }
            // Make it impossible for the sender to re-claim
            // the same deposit.
            bidToCheck.blindedBid = bytes32(0);
        }
        msg.sender.transfer(refund);
    }

    // This is an "internal" function which means that it
    // can only be called from the contract itself (or from
    // derived contracts).
    function placeBid(address bidder, uint value) internal
            returns (bool success)
    {
        if (value <= highestBid) {
            return false;
        }
        if (highestBidder != address(0)) {
            // Refund the previously highest bidder.
            pendingReturns[highestBidder] += highestBid;
        }
        highestBid = value;
        highestBidder = bidder;
        return true;
    }

    /// Withdraw a bid that was overbid.
    function withdraw() public {
        uint amount = pendingReturns[msg.sender];
        if (amount > 0) {
            // It is important to set this to zero because the recipient
            // can call this function again as part of the receiving call
```

```
124             // before `transfer` returns (see the remark above about
125             // conditions -> effects -> interaction).
126             pendingReturns[msg.sender] = 0;
127 
128             msg.sender.transfer(amount);
129         }
130     }
131 
132     /// End the auction and send the highest bid
133     /// to the beneficiary.
134     function auctionEnd()
135         public
136         onlyAfter(revealEnd)
137     {
138         require(!ended);
139         emit AuctionEnded(highestBidder, highestBid);
140         ended = true;
141         beneficiary.transfer(highestBid);
142     }
143 }
```

7.2 Vyper Examples

The examples are from the official Vyper documentation[7].

7.2.1 Simple Open Auction

In the Simple Open Auction example, participants can submit bids during a limited time period. When the auction period ends, a predetermined beneficiary will receive the amount of the highest bid.

```
# Open Auction
beneficiary: public(address)
auction_start: public(timestamp)
auction_end: public(timestamp)

# Current state of auction
highest_bidder: public(address)
highest_bid: public(wei_value)

# Set to true at the end, disallows any change
ended: public(bool)

@public
def __init__(_beneficiary: address, _bidding_time: timedelta):
    self.beneficiary = _beneficiary
    self.auction_start = block.timestamp
    self.auction_end = self.auction_start + _bidding_time

@public
@payable
def bid():
    # Check if bidding period is over.
    assert block.timestamp < self.auction_end
    # Check if bid is high enough
    assert msg.value > self.highest_bid
    if not self.highest_bid == 0:
        # Sends money back to the previous highest bidder
        send(self.highest_bidder, self.highest_bid)
    self.highest_bidder = msg.sender
    self.highest_bid = msg.value
```

```
31
32
33  @public
34  def end_auction():
35      # 1. Conditions
36      # Check if auction endtime has been reached
37      assert block.timestamp >= self.auction_end
38      # Check if this function has already been called
39      assert not self.ended
40
41      # 2. Effects
42      self.ended = True
43
44      # 3. Interaction
45      send(self.beneficiary, self.highest_bid)
```

7.2.2 Safe Remote Purchases

The following example shows an escrow contract where a buyer and a seller can make transactions without a middleman (trusted 3rd party).

```
1   value: public(wei_value) #Value of the item
2   seller: public(address)
3   buyer: public(address)
4   unlocked: public(bool)
5   #@constant
6   #def unlocked() -> bool: #Is a refund possible for the seller?
7   #    return (self.balance == self.value*2)
8
9   @public
10  @payable
11  def __init__():
12      assert (msg.value % 2) == 0
13      self.value = msg.value / 2  #The seller initializes the contract by
14          #posting a safety deposit of 2*value of the item up for sale.
15      self.seller = msg.sender
16      self.unlocked = True
17
18  @public
19  def abort():
```

```
    assert self.unlocked #Is the contract still refundable?
    assert msg.sender == self.seller #Only the seller can refund
        # his deposit before any buyer purchases the item.
    selfdestruct(self.seller) #Refunds the seller and deletes the contract.

@public
@payable
def purchase():
    assert self.unlocked #Is the contract still open (is the item still up for sale)?
    assert msg.value == (2 * self.value) #Is the deposit the correct value?
    self.buyer = msg.sender
    self.unlocked = False

@public
def received():
    assert not self.unlocked #Is the item already purchased and pending confirmation
        # from the buyer?
    assert msg.sender == self.buyer
    send(self.buyer, self.value) #Return the buyer's deposit (=value) to the buyer.
    selfdestruct(self.seller) #Return the seller's deposit (=2*value)
        # and the purchase price (=value) to the seller.
```

7.2.3 CrowdFund

In the CrowdFund contract, participants can contribute to a campaign. If predetermined funding goal is reached, the funds will be sent to the beneficiary. Otherwise, participants will be refunded.

```
# Setup private variables (only callable from within the contract)
funders: {sender: address, value: wei_value}[int128]
nextFunderIndex: int128
beneficiary: address
deadline: timestamp
goal: wei_value
refundIndex: int128
timelimit: timedelta
```

```
@public
def __init__(_beneficiary: address, _goal: wei_value, _timelimit: timedelta):
    self.beneficiary = _beneficiary
    self.deadline = block.timestamp + _timelimit
    self.timelimit = _timelimit
    self.goal = _goal


# Participate in this crowdfunding campaign
@public
@payable
def participate():
    assert block.timestamp < self.deadline

    nfi: int128 = self.nextFunderIndex

    self.funders[nfi] = {sender: msg.sender, value: msg.value}
    self.nextFunderIndex = nfi + 1


# Enough money was raised! Send funds to the beneficiary
@public
def finalize():
    assert block.timestamp >= self.deadline and self.balance >= self.goal

    selfdestruct(self.beneficiary)


# Not enough money was raised! Refund everyone (max 30 people at a time
# to avoid gas limit issues)
@public
def refund():
    assert block.timestamp >= self.deadline and self.balance < self.goal

    ind: int128 = self.refundIndex

    for i in range(ind, ind + 30):
        if i >= self.nextFunderIndex:
            self.refundIndex = self.nextFunderIndex
            return

        send(self.funders[i].sender, self.funders[i].value)
```

```
54          self.funders[i] = None
55
56      self.refundIndex = ind + 30
```

7.3 Scilla Examples

The example below is taken from the Scilla documentation. [4]

7.3.1 Hello World

```
(* HelloWorld contract *)


(***************************************************)
(*               Associated library                *)
(***************************************************)
library HelloWorld

let one_msg =
  fun (msg : Message) =>
  let nil_msg = Nil {Message} in
  Cons {Message} msg nil_msg

let not_owner_code  = Int32 1
let set_hello_code  = Int32 2

(***************************************************)
(*             The contract definition             *)
(***************************************************)

contract HelloWorld
(owner: ByStr20)

field welcome_msg : String = ""

transition setHello (msg : String)
  is_owner = builtin eq owner _sender;
  match is_owner with
  | False =>
    msg = {_tag : "Main"; _recipient : _sender; _amount : 0; code : not_owner_code};
    msgs = one_msg msg;
    send msgs
  | True =>
    welcome_msg := msg;
```

```
35      msg = {_tag : "Main"; _recipient : _sender; _amount : 0; code : set_hello_code};
36      msgs = one_msg msg;
37      send msgs
38    end
39  end
40
41  transition getHello ()
42      r <- welcome_msg;
43      msg = {_tag : Main; _recipient : _sender; _amount : 0; msg : r};
44      msgs = one_msg msg;
45      send msgs
46  end
```

Bibliography

[1] Amrit Kumar Ilya Sergey and Aquinas Hobor. *Scilla Whitepaper*. URL: https://arxiv.org/pdf/1801.00687.pdf. (accessed: 30.09.2018).

[2] Investopedia. *Smart Contracts Definition*. URL: https://www.investopedia.com/terms/s/smart-contracts.asp. (accessed: 27.11.2018).

[3] Amrit Kumar. *Scilla Whitepaper*. URL: https://blog.zilliqa.com/scilla-design-story-piece-by-piece-part-1-why-do-we-need-a-new-language-27d5f14ae661. (accessed: 30.09.2018).

[4] Zilliqa Research. *Scilla Readthedocs*. URL: https://scilla.readthedocs.io/en/latest/index.html. (accessed: 04.09.2018).

[5] Scilla. *Scilla Lang*. URL: https://scilla-lang.org/. (accessed: 30.09.2018).

[6] Solidity. *Solidity 0.4.25 documentation*. URL: https://solidity.readthedocs.io/en/v0.4.25/. (accessed: 26.09.2018).

[7] Vyper. *Vyper 0.1.0-beta3 documentation*. URL: https://vyper.readthedocs.io/en/latest/index.html. (accessed: 26.09.2018).

Appendix C

Generated EBNF

The following EBNF grammar is generated from the ANTLR grammar using the online tool http://bottlecaps.de/convert/ on 5.12.2018.

```
/* converted on Wed Dec 5, 2018, 14:59 (UTC+01) by antlr_4-to-w3c v0.45
which is Copyright (c) 2011-2018 by Gunther Rademacher <grd@gmx.net> */

program  ::= NLS* versionDirective interfaceDeclaration* contractDeclaration EOF
versionDirective
         ::= 'version' INTEGER '.' INTEGER NLS
interfaceDeclaration
         ::= 'interface' IDENTIFIER '{' NLS* interfacePart* '}' NLS
interfacePart
         ::= functionSignature NLS
functionSignature
         ::= annotation* ( type | '(' type ( ',' type )* ')' )
         IDENTIFIER '(' paramList? ')'
contractDeclaration
         ::= 'contract' IDENTIFIER ( 'is' IDENTIFIER ( ',' IDENTIFIER )* )?
         '{' ( NLS | contractPart )* '}' NLS?
contractPart
         ::= variableDeclaration
           | structDeclaration
           | errorDeclaration
           | enumDeclaration
           | eventDeclaration
           | constructorDeclaration
           | functionDeclaration
```

```
variableDeclaration
        ::= 'readonly'? type IDENTIFIER assignment? NLS
structDeclaration
        ::= 'struct' IDENTIFIER '{' NLS* structField* '}' NLS
errorDeclaration
        ::= 'error' IDENTIFIER '{' NLS* structField* '}' NLS
structField
        ::= type IDENTIFIER NLS
eventDeclaration
        ::= 'event' IDENTIFIER '(' paramList? ')' NLS
enumDeclaration
        ::= 'enum' IDENTIFIER '{' NLS* IDENTIFIER
        ( ',' NLS* IDENTIFIER )* NLS* '}' NLS
constructorDeclaration
        ::= annotation* 'constructor' '(' paramList? ')' statementBlock
functionDeclaration
        ::= annotation* functionHead statementBlock
functionHead
        ::= 'internal'? 'function' ( type | '(' type ( ',' type )* ')' )
        IDENTIFIER '(' paramList? ')'
annotation
        ::= '[' IDENTIFIER ( ':' expression )? ']' NLS
paramList
        ::= parameter ( ',' parameter )* ( ',' defaultParameter )*
parameter
        ::= type IDENTIFIER
defaultParameter
        ::= parameter assignment
type    ::= arrayType
          | mapType
          | IDENTIFIER
arrayType
        ::= IDENTIFIER '[' ']'
mapType ::= 'Map' '<' type ',' type '>'
statementBlock
        ::= '{' ( NLS | statement )* '}'
statement
        ::= assignmentStatement
          | returnStatement
          | expressionStatement
          | sendStatement
          | emitStatement
```

```
            | variableDeclaration
            | ifStatement
            | forEachStatement
            | forStatement
            | mapForEachStatement
            | breakStatement
            | continueStatement
            | throwStatement
emitStatement
         ::= 'emit' expression NLS
deleteStatement
         ::= 'delete' expression NLS
ifStatement
         ::= 'if' '(' expression ')' statementBlock
         ( 'else if' '(' expression ')' statementBlock )?
         ( 'else' statementBlock )?
forStatement
         ::= 'for' '(' IDENTIFIER ':' rangeStatement ')' statementBlock
forEachStatement
         ::= 'foreach' '(' ( IDENTIFIER ',' )?
         type IDENTIFIER ':' expression ')' statementBlock
mapForEachStatement
         ::= 'foreach' '(' type IDENTIFIER ','
         type IDENTIFIER ':' expression ')' statementBlock
breakStatement
         ::= 'break' NLS
continueStatement
         ::= 'continue' NLS
rangeStatement
         ::= expression? 'to' expression ( 'by' expression )?
expressionStatement
         ::= expression NLS
sendStatement
         ::= expression '.' 'send' '(' expression? ')' NLS
argumentList
         ::= ( expression ( ',' expression )* | namedArgument )
         ( ',' namedArgument )*
namedArgument
         ::= IDENTIFIER '=' expression
assignmentStatement
         ::= expression assignment NLS
assignment
```

```
109         ::= '=' expression
110 designator
111         ::= IDENTIFIER
112 throwStatement
113         ::= 'throw' IDENTIFIER '{' argumentList? '}' NLS
114 returnStatement
115         ::= 'return' ( expression ( ',' expression )* )? NLS
116 expression
117         ::= expression ( '++' | '--' | '[' expression ']'
118         | '.' IDENTIFIER | '(' argumentList? ')'
119         | ( ( '+' | '-' | '**' | '*' | '/' | '\%' | '<<' | '>>' | '&' | '^'
120         | '|' ) '='? | '<' | '>' | '<=' | '>=' | '==' | '!=' | '&&' | '||'
121         | '?' expression ':' ) expression )
122         | newCreation
123         | '(' ( expression ')' | type ')' expression )
124         | ( '++' | '--' | '+' | '-' | '!' | TILDE ) expression
125         | operand
126 newCreation
127         ::= structCreation
128           | arrayCreation
129           | mapCreation
130 structCreation
131         ::= 'new' IDENTIFIER '(' argumentList? ')'
132 arrayCreation
133         ::= 'new' IDENTIFIER '[' ( expression ']' ( '{' '}' )? | ']'
134         '{' expression ( ',' expression )* '}' )
135 mapCreation
136         ::= 'new' mapType '(' ')'
137 operand  ::= literal
138           | designator
139 literal  ::= INTEGER
140           | CHARACTER
141           | STRING
142           | BOOL
143 _        ::= WHITE_SPACE
144           | LINE_COMMENT
145           | BLOCK_COMMENT
146          /* ws: definition */
147
148 <?TOKENS?>
149
150 BOOL     ::= 'true'
```

```
            | 'false'
TILDE     ::= #x007e
IDENTIFIER
          ::= ( '_' | ALPHA_LETTER ) ( '_' | ALPHA_LETTER | DEC_DIGIT )*
ALPHA_LETTER
          ::= [a-zA-Z]
INTEGER   ::= DEC_DIGIT_LIT
            | HEX_DIGIT_LIT
HEX_DIGIT_LIT
          ::= '0x' HEX_DIGIT+
HEX_DIGIT
          ::= [0-9a-fA-F]
DEC_DIGIT_LIT
          ::= DEC_DIGIT+
DEC_DIGIT
          ::= [0-9]
STRING    ::= '"' UNICODE_CHAR* '"'
CHARACTER
          ::= "'" ( ESCAPED_CHAR | UNICODE_CHAR ) "'"
ESCAPED_CHAR
          ::= '\' ( '0' | 'n' | '\' | "'" | '"' )
UNICODE_CHAR
          ::= [^#xd#xa]
NLS       ::= NL+
NL        ::= [#xa#xd]
WHITE_SPACE
          ::= [ #x9#xc#xd]+
LINE_COMMENT
          ::= '//' [^#xd#xa]*
BLOCK_COMMENT?
          ::= '/*' .* '*/'
EOF       ::= $
```

Appendix D

GitHub Repository

Lazo Specification Git Repository