ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

**Scuola di Scienze**
**Corso di Laurea in Informatica**

# Gas Analysis in Vyper Extension for the Ethereum Blockchain

**Relatore:**
**Prof. Ugo Dal Lago**

**Presentata da:**
**Michele Luca Contalbo**

Anno Accademico 2019/2020

*Alla famiglia, agli amici, a coloro che mi hanno permesso di concludere questo percorso...*

# Contents

**Introduction** **iii**

**1 Background** **1**

1.1 Blockchain . . . . . . . . . . 1

1.2 Block . . . . . . . . . . 2

1.3 Ethereum . . . . . . . . . . 4

1.3.1 Transactions . . . . . . . . . . 4

1.3.2 Payments and currency . . . . . . . . . . 5

1.4 Vyper . . . . . . . . . . 5

1.4.1 State Variables . . . . . . . . . . 6

1.4.2 Types . . . . . . . . . . 6

1.4.3 Environment Variables . . . . . . . . . . 7

1.4.4 Functions . . . . . . . . . . 7

**2 Rewrite Systems** **9**

2.1 Integer Term Rewrite Systems . . . . . . . . . . 10

2.1.1 Definition and syntax . . . . . . . . . . 10

2.1.2 Complexity ITRS . . . . . . . . . . 13

2.1.3 Termination . . . . . . . . . . 14

2.2 KoAT . . . . . . . . . . 15

2.2.1 Conditional CITRS . . . . . . . . . . 16

2.2.2 The Com_k command . . . . . . . . . . 17

2.2.3 Structure of KoAT programs . . . . . . . . . . 17

**3 Calculus Semantics** **21**
3.1 Control Flow Graph . . . 21
3.2 Operational Semantics . . . 25
3.2.1 Transition Systems . . . 25
3.2.2 Auxiliary Functions . . . 25
3.2.3 Semantics of commands . . . 26
3.2.4 Error semantics . . . 31
3.2.5 Considerations about the semantics of commands . . . 32

**4 Gas Calculation** **33**
4.1 Operational codes . . . 33
4.2 The Ethereum paradigm . . . 34
4.3 Compilers . . . 37
4.3.1 Vyper compiler . . . 39

**5 Testing** **43**
5.1 Fibonacci algorithm . . . 43
5.2 Insertion sort . . . 45
5.3 Shell sort . . . 45
5.4 Binary search . . . 46

**Conclusions** **49**

**Bibliography** **51**

# Introduction

In the last years, blockchain technologies like Bitcoin have demonstrated that it is possible to build decentralized, anonymous and secure systems to make digital transactions. These technologies are becoming very popular and are being applied in a lot of fields. One of these fields is finance (with online payments) but we can also find blockchains for data sharing, Internet of Things or charity donations. In 2017, the University of Edinburgh managed to create the first full autonomous corporation, called BitBarista. This is a coffee machine prototype and it interacts with customers by exchanging Bitcoins. As instance, the customer pays in Bitcoin for the coffee, but he could also receive money from BitBarista, in exchange for maintainance.
In 2013, a cryptocurrency researcher and programmer called Vitalik Buterin saw the potential of blockchains and wondered whether it was possible to create something that extends the functionalities of Bitcoin. Eventually, he proposed Ethereum, which is a decentralized open source blockchain featuring smart contract functionality. This feature is one of the main differences between Bitcoin and Ethereum, and it highlights how Ethereum is focused on facilitating the creation of peer-to-peer contracts and applications. At the same time, Ethereum has its own currency called Ether (ETH) which is like Bitcoin and it can be used to instantly send money to anyone in the world. Ethereum is gaining more and more popularity since it allows programmers to potentially create decentralized organizations. Moreover, applications in Ethereum cannot be censored, since the programs are not deployed in only one web server, but accross the whole blockchain. To each instruction of the EVM (Ethereum Virtual Machine) bytecode, corresponds a certain gas usage, which is the unit of measure of the bytecode's complexity. Thus, one of the main problems of the Ethereum blockchain is to establish upper bounds to the

operational cost of certain operations. In fact, users who want to benefit from a certain dApp (decentralized application, also called smart contract in Ethereum) must pay an amount of Ether to the miner for their effort and that amount is equal to the total amount of gas it took to complete the operation. Upper bounds could also be used as a unit of measure between different dApps. In other words, upper bounds could show, for a certain task, which could be the best algorithm (at least regarding the cost of the execution). A lot of researches have been conducted to obtain more precise bounds at a less computational effort, but, in case of while loops or recursion, such bounds cannot always be computed. For this reason, a non Turing-Complete language was created, called Vyper, so that, since it doesn't allow programs to diverge, it is possible to compute precise and non symbolic upper bounds. Vyper is becoming more and more popular even because it is derived from Python, but since it is not Turing-Complete, a lot of programmers still prefer other alternatives, like Solidity. In this thesis, we consider extensions of the Vyper language and compiler, allowing the programmer to use while constructs in smart contracts. Afterward, we will illustrate the implementation of a compiler from Vyper source code to CITRS (Complexity Integer Term Rewrite System). Then, a tool named KoAT[24] will be used to infer symbolic upper bounds on while loops from the CITRS code, which will allow us to calculate the maximum amount of gas spent by a program.

# Related Work

A lot of research have been conducted regarding Vyper. Mavridou et al.[8,9] highlight the main security differences between Vyper and Solidity, which is a Javascript-based Ethereum programming language. From the comparison, they claim that Vyper manages to fully address 3 vulnerabilities that are present in Solidity, which are:

- Integer overflow and underflow: the contract execution reverts if they are detected;
- Denial of Service with unbounded operations: occurs when the operations required in the execution of a function exceed the block gas limit. As said in the previous chapter, Vyper does not have this problem, since upper bounds can be precisely calculated;
- Unchecked call return value: due to Solidity's discrepancy in handling exceptions occured in called contracts.

Piper Merriam, in an article posted on the 8th of January 2020 [10], has said that, even though Vyper has been approved by Vitalik Buterin as a valid replacement for Serpent (its python predecessor), there are still a few bugs in the compiler that make it still unsafe to use. Maintaners are working to fix these errors, but it may take a lot of time.

Robin Sierra has introduced 2Vyper[11], which is a verifier for smart contracts that can prove the absence of various security problems, like success or failures of a function or unpredictable state. In the research, the specifications were encoded in an intermediate language called Viper that, eventually, gets checked by an SMT solver.

Regarding term rewrite systems, some of the most important works are [1,2,3,4]. J.W.Klop[1] introduces several basic concepts regarding rewrite systems, including ab-

stract rewrite systems and the Knuth-Bendix completion, which is an algorithm to translate sets of equations into equivalent term rewrite systems. N.Dershowitz et Mitsuhiro Okada[2] provide a proof theoretic approach into term rewrite systems. With this method, they analyze important properties of term rewrite systems, like termination and confluence. Dougherty et al.[3] have presented a formalism called "Addressed Term Rewriting Systems", which can be used to define the operational semantics of programming languages.

# Chapter 1

# Background

This chapter gives a background on the topics of the thesis. It will show how Ethereum works and some of the main features of Vyper. For formal definitions, we suggest to check the Ethereum Yellow Paper[4] and the Vyper documentation[5].

## 1.1 Blockchain

A blockchain is, as the name may suggest, an always growing serie of blocks that are timestamped. It can also be considered as a database of all the transactions and operations deployed into the network. Indeed, every block is needed to mine new blocks, thus, to preserve the blockchain's integrity, no block can be eventually deleted. The three main pillars that allow each blockchain to be secure and scalable are:

- **Decentralization**: there is no third-party organization that manages and controls the blockchain;
- **Transparency**: each node is able to control the whole state of the blockchain;
- **Immutability**: even if technically possible, it is almost infeasible to change a validated block's data.

Encryption algorithms provide privacy and security of information. Indeed, in blockchains, they can encrypt financial records and payments, making it impossible to read the information for someone that does not have the decryption key. The blockchain does not

define a particular encryption algorithm to be used for making safe transactions. Nevertheless, encryption algorithms or hashing functions like AES and SHA are usually chosen. In general, we could say that encryption provides safety properties for cryptocurrencies, but it is the structure of the blockchain that makes "digital money" usable. As instance, euros are regulated and verified by a central authority (usually banks). This authority manages the savings of a high number of customers, thus, if somehow it collapses, the users savings could be at risk. Instead, in blockchains, there is no need for a central authority, since the operations are spread accross the network. This reduces the overall risk. As mentioned in the introduction, two of the most famous blockchains are Bitcoin and Ethereum. They have some similarities: they are both public, have their own cryptocurrencies but also a Proof of Work mining (PoW), meaning that the miner must prove that the mining process had a certain computational effort. At the same time, we could say that Ethereum extends the blockchain concepts from Bitcoin:

- it allows to run code equivalently on many computers (smart contracts);
- it has also proposed the Proof of Stack (instead of PoW). In PoS, the creator of the next block is generally chosen through a random process.

## 1.2 Block

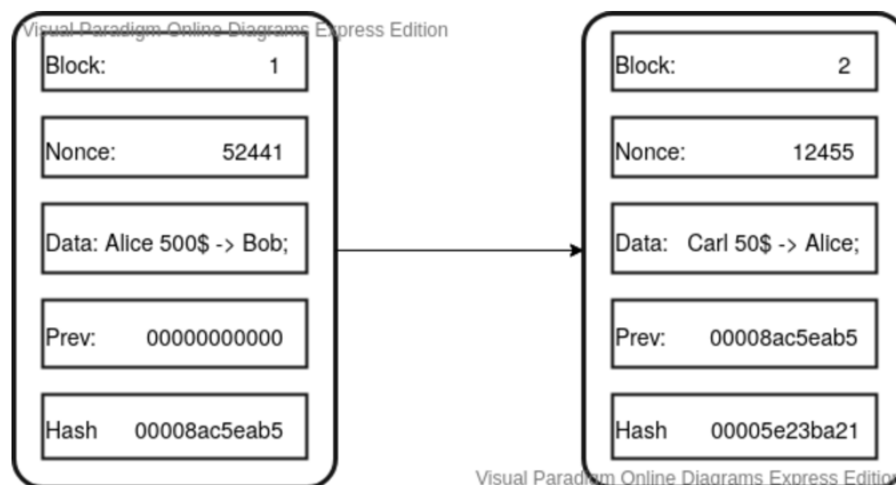
A block inside a blockchain is composed of the following parts:

- a unique block number;
- transaction data, which indicates the outcome of the execution of a certain function;
- the hash of the previous block;
- the nonce, which is useful to ensure the security of the blockchain.

When a block has to be validated, the mining process starts, with several computers competing with each other to mine the block. These computers must find a suitable nonce (number) such that the hash of the block with the nonce satisfies a certain condition.

The condition is changed quite often to control the mining ratio of the blockchain: if too many blocks are being mined, then the condition is made harder and, on the opposite situation, it is made easier. As an example, the condition could be that each hash has to contain a certain substring. When a suitable nonce is found, then the block has to be validated by each node (computer) of the blockchain. This can be considered as a poll and, if more than half of the votes are positive, then the block is added to the blockchain. Now let us consider an user who wants to change the transaction data of an already validated block. The user must recalculate the nonce (so, the hash) of the block that has been changed, but also the hashes of all the next blocks, since they are calculated based on the previous values. So, while it is technically possible to change the value of a block, it is infeasible to recalculate all the next values.

**Example 1.2.1.** Let us consider an example, taken from [7]. We have 2 blocks in the blockchain:




When mining a block, we have to find a suitable nonce. Hence, at the beginning of the process, the `nonce` value is unknown. The value of `data`, instead, depends on the operations done by the users. The `prev` entry is always equal to the hash of the previous block, while the `hash` is the hashed value of the entire block (calculated when finding the nonce). In this example, the condition that the nonce satisfies could be the 4 leading zeros in the hash value. Since `hash` is obtained by hashing the whole block (including `prev`, i.e. a value dependent from the previous block), we can see that, if

block 1 is modified, then also the hash values of the next blocks must be changed (and their nonces too).

## 1.3 Ethereum

Ethereum is a blockchain that aims to facilitate transactions without any third-party organization that controls it. This is obtained because every agreement or contract is enforced autonomously through a rich and unambiguous language (EVM bytecode). In other words, there is no "in the middle" organization that receives and stores data, so every user has complete power over his own information. Every smart contract is open source, hence every user can directly read what the code does. Smart contracts are also immutable, since once a contract is deployed in the network, it stays in the network. Indeed, as previously said, the blockchain stores every block and they cannot be modified. Thus, a smart contract and its associated address (from where the dApp can be accessed) cannot change, since this would bring us to change old blocks.

### 1.3.1 Transactions

A transaction is a cryptographically-signed instruction used for message calls or for the creation of new accounts with associated code. In both cases, the transaction specifies a number of common fields. The most important fields in this research will be:

- `gasPrice`: it is a value equal to the number of `Wei` to be paid to the miner per unit of gas;
- `gasLimit`: it is a value equal to the maximum amount of gas that can be paid for the transaction. If it is not sufficient for the termination of the operation, the gas will still be paid and the transaction aborted;
- `value`: it is a value equal to the number of `Wei` to be transferred to the message call's recipient.

### 1.3.2 Payments and currency

Ether is a fundamental component in Ethereum and it not only allows to make normal payments, but it is also the fuel of the entire blockchain. Through the entire research, any reference to the value of Ether should be counted in Wei, whose value, together with other units, is described in Table 1.1.

| Unit | Wei value | Wei |
|---|---|---|
| wei | 1 wei | 1 |
| Kwei | 1e3 wei | 1000 |
| Mwei | 1e6 wei | 1000000 |
| Gwei | 1e9 wei | 1000000000 |
| microether | 1e12 wei | 1000000000000 |
| milliether | 1e15 wei | 1000000000000000 |
| ether | 1e18 wei | 1000000000000000000 |

Table 1.1: Ether conversion table

Ethereum Gas, instead, is the amount of Wei to be paid to a miner for the execution of a function. As stated before, every EVM bytecode instruction has a certain cost, which is specified in Appendix G of [4]. In general, the amount of gas to be paid is calculated with

$$G_{total} = G_{price} * G_{used}$$

## 1.4 Vyper

Vyper is a strongly-typed contract-oriented programming language that targets the Ethereum Virtual Machine. Vyper strives to create secure smart contracts, for this reason it does not provide the `while` command nor recursion. In this way, it is possible to check bounds and overflow on arrays. Moreover, it is possible to compute a precise upper bound for the gas consumption of any Vyper function call.
This section will provide basic examples to understand the structure of a Vyper contract.

### 1.4.1 State Variables

State variables are values which are permanently stored in the contract's storage. So, these variables have a global scope and they can be accessed from any function of the contract. We can see an example in listing 1.1.

```
beneficiary: public(address)
auctionStart: public(uint256)
auctionEnd: public(uint256)

# Current state of auction
highestBidder: public(address)
highestBid: public(uint256)

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

# Keep track of refunded bids so we can follow the withdraw pattern
pendingReturns: public(HashMap[address, uint256])
```

Code Listing 1.1: State Variables

The keyword `public` has the same meaning as "public" in an object oriented language. In other words, it is possible to access the variable from outside the contract. Plus, in Vyper, every `public` variable has a getter function already defined. So, `contract_name.beneficiary()` evaluates the value of the corresponding variable.

### 1.4.2 Types

Vyper is a statically typed language, so the type of every variable must be known at compile time. It does not support type inference, so every variable type must be specifically declared, in order for the compiler to work properly. This is a form of manifest typing, since information on the type is not inferred, but it is explicit in the source code. Vyper has the standard boolean and signed or unsigned integers/decimal operators (with their corresponding bit representations). Vyper also has byte arrays, strings, lists and

structs. Some particular types are addresses, which indicated the location of an account, or mappings, which are hash tables. For an example on how to define these types, see Example 1.1.

### 1.4.3 Environment Variables

Environment variables always exist in the namespace and they give information about the blockchain and the current transaction. The most important ones are:

- `block.timestamp`: the UNIX timestamp of the current block;
- `msg.sender`: address of the account who sent the message;
- `msg.value`: Wei sent with the message;
- `msg.gas`: remaining gas;
- `self`: used to reference a contract within itself.

### 1.4.4 Functions

Functions may only be declared within a contract scope. Each function must have exactly one visibility decorator, plus other optional decorators. Let us see an example:

```
@internal
def _times_two(amount: uint256) -> uint256:
    return amount * 2

@external
def calculate(amount: uint256) -> uint256:
    return self._times_two(amount)

@pure
@external
def pure():
  # this function cannot read state or environment variables
  ...
```

```
@view
@external
def readonly():
    # this function cannot write to state
    ...

@nonpayable
@external
def dont_send_money():
  # this function cannot receive Ether, but can read and write to state
  ...

@external
@nonreentrant("lock")
def make_a_call(_addr: address):
    # this function is protected from re-entrancy
    ...

@payable
@external
def send_me_money():
    # this function can receive ether
    ...
```

Code Listing 1.2: Function Decorators

Functions with the `@internal` decorator are only callable from within the contract, while the `@external` ones also from outside. The `@payable` function is the only one that can receive Ether. So, for instance, a function that transfers money must have that decorator. Instead, the `@nonreentrant` decorator is used to control concurrency and to avoid that function calls work on a non stable contract state.
There are also some built-in functions. The one that is used the most is `send(address, uint256)`, which sends ether to the specified ethereum address.

# Chapter 2

# Rewrite Systems

This chapter focuses on rewrite systems, which are a set of rewriting rules composed by a left term, a right term and a transition function ($\rightarrow$). Given a formula, a rewrite system can be used to replace subterms of the formula with other terms. In other words, if the subterm matches a given left term, then it can be substituted with its right term. Rewrite systems provide a paradigm of computation with straightforward syntax and semantics. By simplifying terms, they ease the automatic verification of intrinsic program properties.

**Example 2.0.1.** The chamelions come in three colors, red, yellow, and green, and wander about continuously. Whenever two chamelions of different colors meet, they both change to the third color. Suppose there are 15 red chamelions, 14 yellow, and 13 green. Can their haphazard meetings lead to a stable state, all sharing the same color?

**Example 2.0.2.** An urn contains 150 black beans and 75 white. Two beans are removed at a time: if they're the same color, a black one is placed in the urn; if they're different, the white one is returned. The process is repeated as long as possible. Is the color of the last bean in the urn predetermined and, if so, what is it?

Both examples give rules to go from a starting state to a final one. We can define the first problem with a set of rules:

$$\begin{aligned} red\ yellow &\rightarrow green\ green \\ green\ yellow &\rightarrow red\ red \\ red\ green &\rightarrow yellow\ yellow \end{aligned}$$

We could try to understand if this set of rules is terminating. As instance, *green green yellow* → *green red red* → *yellow yellow red* → *yellow green green*, which could be rearranged as *green green yellow*, thus the rules are not terminating. We can also define the second problem with a set of rules:

$$\begin{aligned} black\ black &\rightarrow black \\ white\ white &\rightarrow black \\ black\ white &\rightarrow white \\ white\ black &\rightarrow white \end{aligned}$$

In this case, the amount of beans is always decreasing, hence the sequence of reductions will terminate. By using this paradigm, we have been able to easily infer non-termination properties of the two systems. It is also possible to get the computational complexity of these programs. KoAT, as instance, is a tool developed for this reason and it uses a particular type of rewrite systems, called Computational Integer Term rewrite systems (CITRS), from which it is able to estimate a computational upper bound.

## 2.1 Integer Term Rewrite Systems

### 2.1.1 Definition and syntax

**Definition 1.** A term denotes an expression which is recursively constructed from constant symbols, variables and function symbols.
We denote the set of terms as $T(\Sigma, V, C)$, with $\Sigma$ the set of signatures, V the set of variables and C the constants.

We will use the Extended Backus-Naur Form (EBNF) to formally define the syntax of ITRSs. We begin by defining the syntax for the variables and constants (respectively V and C).

$$\langle var \rangle ::= \langle name \rangle$$

$$\langle const \rangle ::= \text{'True'} \mid \text{'False'} \mid \langle digit \rangle \langle digit \rangle^*$$

$$\langle arg \rangle ::= \langle var \rangle \mid \langle const \rangle$$

With the * symbol, we refer to the term repetition (i.e. Kleene's star). We use $\langle name \rangle$ to refer to the set of names the variable may have. In a similar fashion, we will use $\langle digit \rangle$ to refer to the numbers 0...9. We avoid writing their reductions, since they are common in every programming language. For the same reason, we will not include the syntax of mathematical and boolean expressions. However, ITRSs allow to have function calls as operands of algebraic expressions. Now we consider the functions:

$$\langle param \rangle ::= (\ \langle arg \rangle \mid \langle expr \rangle\ )\ (\ \text{','}\ (\ \langle arg \rangle \mid \langle expr \rangle\ )\ )^*$$

$$\langle args \rangle ::= \text{'('}\ \langle param \rangle\ \text{')'}$$

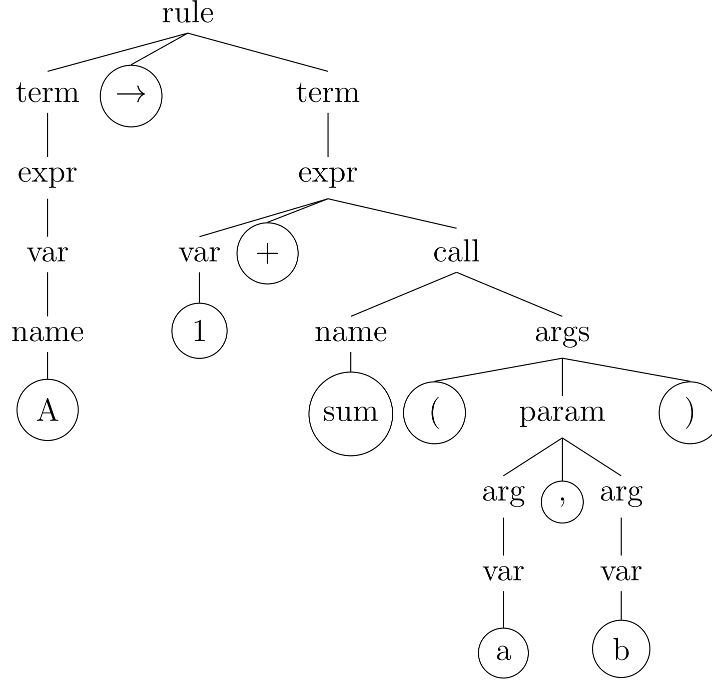
$$\langle call \rangle ::= \langle name \rangle \langle args \rangle$$

**Definition 2.** Let $l, r \in T(\Sigma, V, C)$. A Term Rewriting Rule is a tuple $(l, r)$ and the associated term rewrite relation is defined as $l \to r$, meaning that the right-term $r$ can substitute the left-term $l$.

$$\langle term \rangle ::= \langle expr \rangle$$

$$\langle rule \rangle ::= \langle term \rangle\ \text{'}\!\to\!\text{'}\ \langle term \rangle$$

**Example 2.1.1.** Consider the rule $A \to 1 + sum(a, b)$. Let us see if it belongs to the syntax, by building its corresponding Abstract Syntax Tree (AST).




**Definition 3.** Let $B = \{true, false\}$, $ArithOp = \{+, -, *, /, \%\}$, $RelOp = \{>, \geq, <, \leq, ==, \neq\}$ and $BoolOp = \{\wedge, \Rightarrow\}$. Let I be a set of Term Rewrite Rules and $D = \mathbb{Z} \cup B \cup ArithOp \cup RelOp \cup BoolOp$. Then, I is an Integer Term Rewrite System if $\forall (l, r) \in I.\ l, r \in T(\Sigma \cup D, V, C)$.

$$\langle I \rangle ::= \langle rule \rangle *$$

As notation, if A is an ITRS, $T_A(\Sigma, V, C)$ will denote the set of terms used in A and $\rightarrow_A$ are the term rewrite relations defined over A.

**Example 2.1.2.** Let us consider a function that sums all the natural numbers between two values a and b. We can write an ITRS such that

$$\begin{aligned}
check(a, b) &\rightarrow dsum(a \neq b, a, b) \\
dsum(\texttt{true}, a, b) &\rightarrow dif(a \geq b, a, b) \\
dif(\texttt{true}, a, b) &\rightarrow b + check(a, b + 1) \\
dif(\texttt{false}, a, b) &\rightarrow a + check(a + 1, b) \\
dsum(\texttt{false}, a, b) &\rightarrow a
\end{aligned}$$

We assign, as instance, the values 3 and 5 to $a$ and $b$ respectively.The ITRS executes as follows:

$$\begin{aligned} check(3,5) &\rightarrow dsum(\texttt{true},3,5) \\ dsum(\texttt{true},3,5) &\rightarrow dif(\texttt{false},3,5) \\ dif(\texttt{false},3,5) &\rightarrow 3+check(4,5) \\ 3+check(4,5) &\rightarrow 3+dsum(\texttt{true},4,5) \\ 3+dsum(\texttt{true},4,5) &\rightarrow 3+dif(\texttt{false},4,5) \\ 3+dif(\texttt{false},4,5) &\rightarrow 3+4+check(5,5) \\ 3+4+check(5,5) &\rightarrow 3+4+dsum(\texttt{false},5,5) \\ 3+4+dsum(\texttt{false},a,b) &\rightarrow 3+4+5 \end{aligned}$$

### 2.1.2 Complexity ITRS

Complexity ITRSs (CITRS) are a special type of rewrite systems that are used for complexity analysis. Hence, CITRSs do not focus on what the system computes, but they focus on how many steps it takes to finish the computation. In other words:

**Definition 4.** Let *cp* be a function that calculates the computational complexity of an ITRS A. Then, a CITRS is a rewrite system that calculates $cp(T_A(\Sigma, V, C), \rightarrow_A)$, where by $T_A$ and $\rightarrow_A$ we refer respectively to the set of terms of A and the reduction rules of A.

The *cp* function is strictly dependent on the maximum sequence $l_i \rightarrow^* l_f$, where $l_i$ and $l_f$ are the initial and final state of the CITRS. One way in which it is possible to compute the complexity of the system is by counting the number of $\rightarrow$ steps in the maximum sequence.

**Definition 5.** The derivation height of a term t with respect to the relation $\rightarrow$ on terms is the length of the longest sequence of $\rightarrow$-steps starting with t, i.e.,

$$dh(t, \rightarrow) = \max\{e \mid \exists t' \in T(\Sigma, V, C).\ t \rightarrow^e t'\}$$

**Definition 6.** Consider an ITRS A. The complexity function is defined as

$$cp(T_A(\Sigma, V, C), \rightarrow_A) = \max\{dh(t, \rightarrow_A) \mid \forall t \in T(\Sigma, V, C)\}$$

### 2.1.3 Termination

We've already partially addressed the topic of termination of rewrite systems in the Example 2.0.1. We've shown that it is possible to compute endless loops in rewrite systems (hence in CITRSs). This, together with other properties of rewrite systems, makes the paradigm turing-complete. For this reason, it is possible to translate programs written in high-level programming languages (like Python or Vyper with the while command) into semantically equivalent CITRSs. There have been studies on proof theoretic tools, which are useful for analyzing the termination property of rewrite systems[2]. This property is strictly correlated to the calculation of computational upper bounds, since failure in proving the termination means that the system may diverge, i.e. not have an upper bound.

**Definition 7.** A rewrite system is *terminating* if there are no in infinite derivations $l1 \rightarrow l2 \rightarrow ...$

Most of the approaches in literature try to prove the termination of programs by defining a certain reduction ordering $<$. In particular, if a rewrite system consists of a set of (finite) rules, then the reduction ordering $<$ is the smallest ordering such that $\forall i \leq n, l_i > r_i$. In other words, $s > t$ if $t$ may be obtained from $s$ by one or more applications of rule reductions given by the rewrite system. To show termination of a system over a set of variable-free terms, it is sufficient to show a well-founded monotonic ordering $\prec$ , such that $\forall i \leq n, l_i \prec r_i$.

**Definition 8.** A well-founded set X is a set where $\forall x, y \in X, x > y\ is\ true \vee x < y\ is\ true \vee x = y\ is\ true$ and every subset of X has a minimal element.

Well-founded sets that may be used to create a relation order are ordinal numbers, which are "labels" that can be assigned to elements of a collection.

**Definition 9.** A set S is an ordinal if and only if S is well-ordered and every element of S is also a subset of S.

**Example 2.1.3.** Let us take the first few Von Neumann ordinals:

$$0 = \{\}$$
$$1 = \{0\}$$
$$2 = \{0, 1\}$$
$$3 = \{0, 1, 2\}$$
$$4 = \{0, 1, 2, 3\}$$

We can see that every natural number is an ordinal. As instance, 3 is an element of 4 and, at the same time, it is equal to the well-founded set $\{0, 1, 2\}$.

It is possible to map each well-founded set S to one of the Von Neumann ordinals, thus using them to define an order relation over S. Generally, in a proof theory tool, it could be possible to assign to each term an ordinal, then create an algorithm to define the reduction steps and demonstrate that, for each reduction step, the relative ordinal numbers decrease (hence, the system stops eventually). This topic is covered in [2] by linking Ackermann ordinals to term rewrite systems. Finally, it can be stated that, by choosing an appropriate reduction ordering, decreasing systems (with respect to the ordering) eventually terminate.

## 2.2 KoAT

KoAT is an automatic complexity analyzer that takes as input a CITRS and outputs an upper bound if termination can be proven. Some modifications to the syntax given in the previous sections are needed in order to be able to use KoAT. The modifications are:

- a switch from CITRS to conditional CITRS;
- left terms must be states and right terms must be function calls. Specifically, right terms are `Com_k()` calls;

- information about the system, like the variables used and the initial state, must be explicitly defined;

When we say that left and right terms must be states or function calls, we mean that we cannot have expressions as in Example 2.1.2, where we had, as instance, terms like $3 + dsum(\texttt{true}, 4, 5)$. Every term must be a state or a function call, where the states, in a CITRS, define a particular configuration of the system. As instance, in Example 2.2.3, the states are `l0`,`l1`,`l2` and `l3`. Function calls, like `Com_k()`, are applied over states. These calls implement the complexity calculation of the system. The rewrite system used in KoAT is a conditional CITRS and metadata about the system must be explicitly defined, like the name of the variables used, the start state, the goal of the system etc.

### 2.2.1 Conditional CITRS

**Definition 10.** Let A be an CITRS. A conditional CITRS is a CITRS where $\forall (l, r) \in A, \exists c \in C.\ l \to_A r \iff c = true$, where $C$ is the set of boolean expressions.

For the sake of readability, the conditional rewrite relation will be written $l \to r : | : c$. This syntax is the one used in KoAT. This extension of the definition of CITRS does not change the power of the formalism. In fact, every CITRS can always be translated into an equivalent conditional ITRS.

**Example 2.2.1.** Consider Example 2.1.2. Let us try to write the same function as a conditional ITRS:

$$\begin{aligned}
check(a, b) &\to dsum1(a, b) : | : a \neq b \\
check(a, b) &\to 0 : | : a == b \\
dsum1(a, b) &\to b + check(a, b + 1) : | : a \geq b \\
dsum2(a, b) &\to a + check(a + 1, b) : | : a < b
\end{aligned}$$

It is easy to check that the 2 systems always produce the same output. At the same time, conditional CITRS allow us to write systems in a more compact manner.

From now on, as notation, we will refer to conditional CITRSs as CITRSs.

### 2.2.2 The Com_k command

The `Com_k()` command is used to model a special type of recursion which leads to transitions having multiple (k-many in case of `Com_k`) target terms.

**Example 2.2.2.** Let us consider a simple program that calculates the factorial of an input x:

```
int fac(int x)
  r := 1
  if x > 0 then
    r := x * fac(x - 1)
  fi
  return r
```

This program could be translated into CITRS, resulting in

$$l0(r,x) \to l1(0,x) : | : true$$
$$l1(r,x) \to \{l1(r=r', x-1), l0(r,x)\} : | : x \geq 0$$
$$l1(r,x) \to l2(r,x) : | : x < 0$$

In line 2, there is a transition with 2 target terms. In this way, it is possible to express recursion, since one target term represents the evaluation of the called function, whereas the other target location represents the context which is executed when returning from the function call. Note that r = r', where r' is an arbitrary value, because KoAT does not take the result of procedure calls into account.

We are not interested into $\to$ having multiple target terms, because we use KoAT to compute upper bounds of CITRS obtained from Vyper code, which does not allow recursion. Thus, we will always use `Com_1`.

### 2.2.3 Structure of KoAT programs

Each KoAT program consists of 4 parts:

- `(GOAL COMPLEXITY)`, which specifies that the purpose of the program is to calculate the complexity of the system;
- `(STARTTERM (FUNCTIONSYMBOLS start))`, with `start` being the initial state;
- `(VAR A,B,C...)`, indicating the list of variables used in the system;
- `(RULES CITRS)`, where the actual rewrite system is specified.

As said in Section 2.2, we must modify the conditional CITRS, in order to create a well written KoAT program. In Example 2.2.3, a well written KoAT CITRS is shown, with all the modifications explained in this section,

**Example 2.2.3.** Let us consider the KoAT conditional CITRS sample given by AProVE[23]:

```
(GOAL COMPLEXITY)

(STARTTERM (FUNCTIONSYMBOLS l0))

(VAR A B C D)

(RULES
  l0(A,B,C,D) -> Com_1(l1(0,B,C,D))
  l1(A,B,C,D) -> Com_1(l1(A + 1,B - 1,C,D)) :|: B >= 1
  l1(A,B,C,D) -> Com_1(l2(A,B,A,D)) :|: 0 >= B
  l2(A,B,C,D) -> Com_1(l3(A,B,C,C)) :|: C >= 1
  l3(A,B,C,D) -> Com_1(l3(A,B,C,D - 1)) :|: D >= 1 && C >= 1
  l3(A,B,C,D) -> Com_1(l2(A,B,C - 1,D)) :|: 0 >= D && C >= 1
)
```

Code Listing 2.1: CITRS example

This program creates a loop over the B variable and, at each iteration, B is decreased while A is increased. Once the loops ends, C is put equal to A. Afterward, a nested loop occurs in the states $l2$ and $l3$ where, as soon as D becomes equal to 0, the value of C decreases and D is put equal to C. Thus, we can say that the complexity of this algorithm should be quadratic. We can prove our deduction by running KoAT over this

conditional CITRS. The resulting upper bound calculated by KoAT is $2B^2 + 5B + 2$, which is indeed quadratic ($\mathrm{O}(B^2)$).

# Chapter 3

# Calculus Semantics

In this chapter, we will explain how our tool translates Vyper code into CITRS. We will show which are the structures used to represent the flow of the programs and, in the end, give the semantics of our tool.

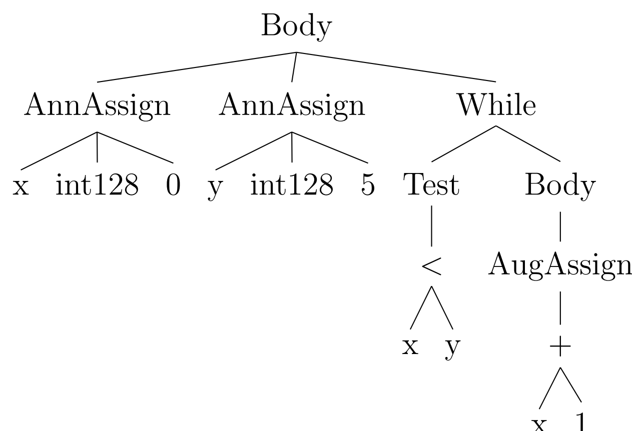
## 3.1 Control Flow Graph

At the beginning of each compilation process, the program's source code gets translated into an Abstract Syntax Tree (AST) which provides a natural representation for the grammatical structure of the source code. This is the process of converting raw data into a tree-based format, which can then be used for further steps of the compilation.

**Example 3.1.1.** Consider the following code:

```
x: int128 = 0
y: int128 = 5

while x<y:
  x += 1
```

Its AST representation is

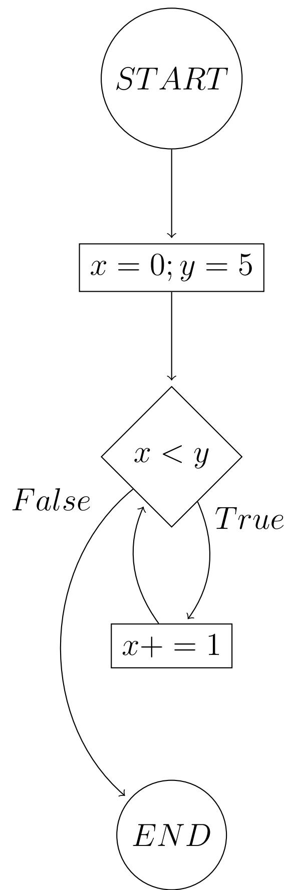



In general, tree structures are useful when providing a grammatical and statical representation of the source code. Anyway, they fail to model program behaviour, thus, they are not used in dynamic analysis of programs. Indeed, ASTs cannot show the flow of the code and the branches the program may take when executing. To get this information, a different structure is more useful and it is the Control Flow Graph (CGF).

The simplest unit of control flow is the basic block, which is the maximal consecutive sequence of branch-free commands (like if, for, while or assert). A basic block always ends at the beginning of a branching instruction or at the end of a loop.

**Definition 11.** A CFG represents the flow between blocks in a program. It is a directed graph $G = (N, E)$ where each node $n \in N$ is a block and each edge $(n_i, n_j) \in E$ passes the control of the flow from the $i$-th node to the $j$-th node.

**Example 3.1.2.** Consider the code in Example 3.1.1.



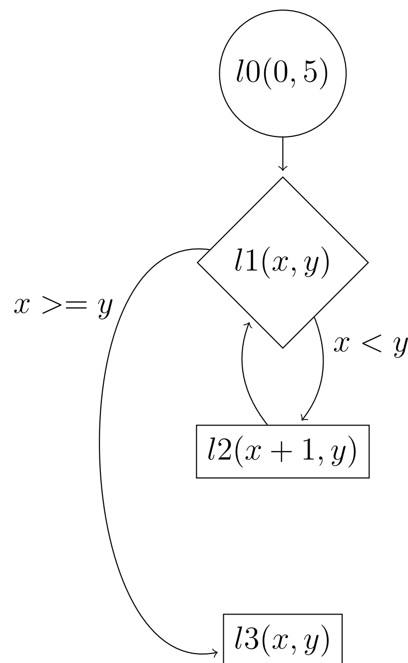

A CITRS can be seen as a CFG where the branching instructions are multiple reductions from a term $l$ to multiple terms $r_1...r_n$. Indeed, the CFG represented in Example 4.1.2, can get written as a CITRS like in the following example:

**Example 3.1.3.** Consider the CFG in Example 3.1.2. Its equivalent CITRS code is the following:

```
(GOAL COMPLEXITY)

(STARTTERM (FUNCTIONSYMBOLS l0))

(VAR x y)

(RULES
  l0(x,y) -> Com_1(l1(0,5))
  l1(x,y) -> Com_1(l2(x,y)) :|: x<y
```

```
  l1(x,y) -> Com_1(l3(x,y)) :|: x>=y
  l2(x,y) -> Com_1(l1(x+1,y))
)
```

We can try to represent the CITRS as a graph




which has the same structure as the CFG shown in Example 3.1.2.

This shows that CITRS code and CFG are strictly correlated. We could follow this approach and translate Vyper programs into CFGs so that, subsequently, we can obtain its relative CITRS. Indeed, each maximal consecutive sequence of non-branching commands is a basic block, and it corresponds to a transition to a new state in the rewrite system. Each branch command can follow 2 edges, depending on whether the condition is met or not. Trivially, in a CITRS, this is represented by a state with 2 possible transitions into 2 different states, with the rules being one the opposite of the other. Also, looping commands have a back-edge targeting a block already passed. In CITRS, this can be obtained by adding a new transition to a previous state.

## 3.2 Operational Semantics

In this section, we define the operational semantics of our calculus. We avoid defining the semantic of boolean and arithmetical commands, already covered in [19]. We will re-use and adapt previously made definitions, to give a mathematical model that describes the programs' behaviour.

### 3.2.1 Transition Systems

There are different approaches to give a semantic to a programming language. The approach that it is used in this work is operational, which consists into building an automaton that, step by step, shows the consequence of the execution of various instructions. In other words, it shows how the calculus works. We use a transition system, which is a tuple $(\delta, \rightarrow)$ where:

- $\delta$ is the set of configurations;
- $\rightarrow \subseteq \delta \times \delta$ and it is defined as the transition relation.

Thus, the transition system is used to describe the system's behaviour based on the transition between configurations. A computation starting from a certain configuration $\delta_0$ is a sequence $\delta_0 \rightarrow \delta_1 \rightarrow \delta_2 \rightarrow ...$ which can be finite or infinite. With $\rightarrow^*$ we define the reflexive and transitive closure of the transition relation $\rightarrow$. This means that, if $\delta_0 \rightarrow \delta_1$ and $\delta_1 \rightarrow \delta_2$, then $\delta_0 \rightarrow^* \delta_2$.

**Definition 12.** The configuration of a transition system $(\delta, \rightarrow)$ is a $\delta = (c, \sigma, G, s)$ where $c$ is a command, $\sigma$ maps variables into their values, $G$ is a CITRS and $s$ is a state of the CITRS.

When the value of a variable $v$ changes into $m$, we use as notation $\sigma[m/v]$.

### 3.2.2 Auxiliary Functions

Here we define the following functions:

- $len(v)$, that returns the length of a list v. If the length of v cannot be statically deduced, it outputs $\infty$;
- $Type(v)$, which outputs the type of the expression v;
- $get_b_state()$, which outputs the name of the state that handles the loop condition;
- $out_block()$, which outputs the name of the state outside the if or while block.
- $else_block()$, outputs the name of the state of the *else* statement.

### 3.2.3 Semantics of commands

Here we present the commands' semantics of the calculus. For readability reasons, when adding new rules to $G$, we avoid specifying how the variables change (i.e. we do not write $s(\sigma)$, but only $s$). However, whenever there is $\rightarrow_f$, we mean that, in that transition, $\sigma$ does not change (the variables remain the same).

$$\frac{}{\langle c, \sigma, G, s\rangle \rightarrow \langle c, \sigma, G, s\rangle} \text{ (Skip)}$$

$$\frac{\langle e, \sigma, G, s\rangle \rightarrow^* m}{\langle v = e, \sigma, G, s\rangle \rightarrow \langle \sigma[v/m], G, s\rangle} \text{ (Assign)}$$

$$\frac{\langle e, \sigma, G, s\rangle \rightarrow^* m \qquad v \in List}{\begin{array}{c}\langle v[i] = e, \sigma, G, s\rangle \rightarrow \langle \sigma[v[i]/m], G \cup \{s \rightarrow s' : | : i = \sigma(i)\}, s'\rangle \\ \forall i.i >= 0 \wedge i < len(v)\end{array}} \text{ (ListAssign)}$$

$$\frac{\langle c_1, \sigma, G, s\rangle \rightarrow \langle c_1', \sigma', G, s\rangle}{\langle c_1; c_2, \sigma, G, s\rangle \rightarrow \langle c_1'; c_2, \sigma', G, s\rangle} \text{ (Seq1)}$$

$$\frac{\langle c_1, \sigma, G, s\rangle \rightarrow \langle \sigma', G, s\rangle}{\langle c_1; c_2, \sigma, G, s\rangle \rightarrow \langle c_2, \sigma', G, s\rangle} \text{ (Seq2)}$$

$$\frac{}{\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, \sigma, G, s\rangle \rightarrow \langle c_1; \texttt{endif}; c_2; \texttt{endelse}, \sigma, G \cup \{s \rightarrow s' : | : e\} \cup \{s \rightarrow s'' : | : \neg e\}, s'\rangle} \text{(If)}$$

In (If) we add to $G$ the rewriting rules for the if-else branch. We add endif and endelse so that we can manage the rules for the end of the if and else statement.

$$\frac{}{\langle \texttt{while}\ e\ \texttt{do}\ c_1; c_2, \sigma, G, s\rangle \rightarrow \langle c_1; \texttt{endwhile}; c_2, \sigma, G \cup \{s \rightarrow s'\} \cup \{s' \rightarrow_f s'' : | : e\} \cup \{s' \rightarrow_f s''' : | : \neg e\}, s''\rangle} \text{(While)}$$

In (While), we add 3 rewriting rules:

- in the first one, we go from state $s$ to state $s'$. This way, we denote how the values of the variables have changed, with respect to initial values at the beginning of state $s$;
- in the second one, we apply the case in which the while condition is satisfied;
- in the third one, we apply the case in which the while condition is not satisfied.

The initial transition is added because, when looping back, we do not go to the state $s$ (which would re-apply commands before the while body) but to $s'$, which is needed only to check whether the condition is met or not.

$$\frac{get_b_state() = s' \qquad out_block() = s''}{\langle \texttt{endwhile}, \sigma, G, s\rangle \rightarrow \langle \sigma, G \cup \{s \rightarrow s'\}, s''\rangle} \text{(EndWhile)}$$

The (EndWhile) reduction adds to $G$ the back-edge rule (the transition back to the beginning of while).

$$\frac{out_block() = s' \qquad else_block() = s''}{\langle \texttt{endif}, \sigma, G, s\rangle \rightarrow \langle \sigma, G \cup \{s \rightarrow s'\}, s''\rangle} \text{ (EndIf)}$$

$$\frac{out_block() = s'}{\langle \texttt{endelse}, \sigma, G, s\rangle \rightarrow \langle \sigma, G \cup \{s \rightarrow s'\}, s'\rangle} \text{ (EndElse)}$$

The `(EndIf)` and `(EndElse)` reduction rules add the rewriting rules for changing the state from the `if` or `else` block to the outer block (i.e. after the `if-else` statement).

**Example 3.2.1.** Consider the following code:

```
x: int128 = 0
y: int128 = 5

while x<y:
  x += 1
```

Let us try to use our tool to get the correspondant CITRS. In the initial state, we have $G = \{\}$ and $s = l0$. In this example, we will use the `;` separator to divide each command of the program. The `;` is not allowed in Vyper, but we will use it, in this case, for readability reasons. Moreover, for the same reason, we will write the content of $G$ only when it changes and we will use some names to indicate some commands in the code ($c1$ for $x : int128 = 5$, $c2$ for $y : int128 = 5$, $e$ for $x < y$, $c3$ for $x+ = 1$). The reduction rules applied are the following:

$$\begin{aligned}
&\langle c1, \{\texttt{Nil}, \texttt{Nil}\}, l0\rangle \rightarrow^{Assign}\\
&\langle \{0, \texttt{Nil}\}, l0\rangle \rightarrow^{Seq2}\\
&\langle c2, \{0, \texttt{Nil}\}, l0\rangle \rightarrow^{Assign}\\
&\langle \{0, 5\}, l0\rangle \rightarrow^{Seq2}\\
&\langle \texttt{while}(e)\ \texttt{do}\ c3, \{0, 5\}, l0\rangle \rightarrow^{While}\\
&\langle c3; \texttt{endwhile}, \{0, 5\}, l1\rangle
\end{aligned}$$

The reduction rule `(While)` changes $G$, so it is a branching instruction. At this point,

$G$ becomes:

$$l0(x,y) \rightarrow l1(0,5)$$
$$l1(x,y) \rightarrow l2(x,y) : | : x < y$$
$$l1(x,y) \rightarrow l3(x,y) : | : x >= y$$

We can continue to apply the reduction rules:

$$\langle c3; \texttt{endwhile}, \{x, y\}, l1\rangle \rightarrow^{Seq2}$$
$$\langle \texttt{endwhile}, \{x+1, y\}, l1\rangle \rightarrow^{EndWhile}$$
$$\langle \{x+1, y\}, l2\rangle$$

In the end, $G$ becomes:

$$l0(x,y) \rightarrow l1(0,5)$$
$$l1(x,y) \rightarrow l2(x,y) : | : x < y$$
$$l1(x,y) \rightarrow l3(x,y) : | : x >= y$$
$$l2(x,y) \rightarrow l1(x+1,y)$$

which, by adding the appropriate metadata (see section 2.2.3), creates our CITRS.

**Example 3.2.2.** Consider the following code:

```
lst: int128[2] = [1,5]
i: int128 = 0

while lst[1] > 0:
  if i%2 == 0:
    lst[i] += 1
  else:
    lst[i] -= 1
```

It is clear that there might be some problems with accessing list elements with symbolic indexes. In CITRS there is no List type, so every element of the list must be considered as a variable on its own, without any other kind of connection to the other elements. The value of the symbolic indexes used to access the list elements are not known at compile time, so it is impossible to know which element is being used.

This problem is solved by `(ListAssign)`. Combinatorially, it adds to the CITRS an amount of Term Rewrite Rules equal to the length of the list with the symbolic index (as instance, sym_ind). The i-th Term Rewrite Rule has the value of $i$ at the place of the symbolic index and has one more condition $sym_ind = i$.

**Example 3.2.3.** Let us reconsider the previous example. We can apply the reductions, and obtain:

```
(GOAL COMPLEXITY)

(STARTTERM (FUNCTIONSYMBOLS l0))

(VAR lst_e0 lst_e1 i)

(RULES
  l0(lst_e0,lst_e1,i) -> Com_1(l1(1,5,0))
  l1(lst_e0,lst_e1,i) -> Com_1(l2(lst_e0,lst_e1,i)) :|: lst_e1 > 0
  l1(lst_e0,lst_e1,i) -> Com_1(l3(lst_e0,lst_e1,i)) :|: lst_e1 <= 0
  l2(lst_e0,lst_e1,i) -> Com_1(l4(lst_e0,lst_e1,i)) :|: i%2 = 0
  l2(lst_e0,lst_e1,i) -> Com_1(l5(lst_e0,lst_e1,i)) :|: i%2 != 0
  l4(lst_e0,lst_e1,i) -> Com_1(l6(lst_e0+1,lst_e1,i)) :|: i = 0
  l4(lst_e0,lst_e1,i) -> Com_1(l6(lst_e0,lst_e1+1,i)) :|: i = 1
  l5(lst_e0,lst_e1,i) -> Com_1(l6(lst_e0-1,lst_e1,i)) :|: i = 0
  l5(lst_e0,lst_e1,i) -> Com_1(l6(lst_e0,lst_e1-1,i)) :|: i = 1
  l6(lst_e0,lst_e1,i) -> Com_1(l1(lst_e0,lst_e1,i))
)
```

As notation, we will use $[arrayname]_e[pos]$ to refer to the pos-th element of the array. In this case, the assignments to list elements are parsed into 2 Term Rewriting Rules (the number depends on the list length) who differ on which element is increased

(or decreased). The conditions, that constrain the value of $i$, make the Term Rewriting Rules mutual exclusive.

This way, we add a set of rules that covers all the possible branches that the program may take during runtime. Even though there may be some transitions that will never be used (as instance, the transition l4 to l6 with i = 1 in the previous example), they will be eventually canceled by KoAT. This reduction makes the algorithm more computationally complex. Indeed, it is easy to see that, for an instruction like $list1[index1] = list2[index2]$, our tool will need to consider every combination of the value of $index1$ and $index2$, which, in total, are $len(list1) * len(list2)$ pairs.

### 3.2.4 Error semantics

$$\frac{Type(v) \neq Int}{\langle v, \sigma, G, s \rangle \rightarrow \texttt{err}} \quad \text{(TypeErr)}$$

$$\frac{isFunc(f) = True}{\langle f, \sigma, G, s \rangle \rightarrow \texttt{err}} \quad \text{(FuncErr)}$$

Trivially:

- we cannot create a CITRS if there are variables which are not integers;
- we cannot create a CITRS if external calls are made.

Thus, the tool throws an error only in 2 cases: when there is a variable which cannot be casted to an integer and when there is a call to an external function. Note that our tool is only an integrative part of the entire compiler, which also contains the code of the standard Vyper compiler. This means that the errors shown above are not the only ones the new compiler can give, since it also includes the native exceptions of Vyper.

### 3.2.5 Considerations about the semantics of commands

We will not give a formal proof of soundness properties for our calculus. Nevertheless, we claim that the CITRS obtained is semantically equivalent to the initial Vyper program. Indeed, as shown in Section 3.1, the produced CITRS has the same structure of the CFG of the source program. This can be seen by using Python native libraries, like `pycfg`, which outputs the CFG of the program. We also claim that the `(Assign)` and `(ListAssign)` reductions are sound, since all the changes of variables' values are taken into account by $\sigma$.

# Chapter 4

# Gas Calculation

In this chapter, we will explain how Ethereum calculates the total amount of runtime gas used and we will present how the Vyper compiler statically estimates an upper bound of the gas consumption (except for `while` commands). This way, we are able to integrate the existing computational analysis tools of Vyper with our tool on non-iterative loops. Thus, we could obtain non-symbolic bounds with non-looping commands and symbolic bounds for `while` looping. For the calculation of the gas consumption, we will not consider every operational code, but only the most used ones. In other words, we avoid giving a detailed explanation of all the operational codes[4], but we choose to follow a more general approach, presenting the most frequent patterns in Ethereum bytecode.

## 4.1 Operational codes

The Ethereum bytecode is composed by operational codes that work on the stack (hereafter referred as $\mu$). Some of these codes are presented in the following table:

| Operational Codes | |
|---|---|
| Mnemonic | Description |
| PUSH[x] | Pushes a x-byte value onto $\mu$ |
| POP | Pops a (u)int256 off $\mu$ and discards it |
| JUMP | Unconditional jump to a destination address taken from $\mu$ |
| JUMPI | Takes a destination and condition from $\mu$. Based on the condition, the execution can jump to the destination address |
| JUMPDEST | Metadata to annotate possible jump destinations |
| MSTORE | Takes an offset and a value from $\mu$ and writes to memory the value |
| MLOAD | Takes an offset and writes the relative value onto $\mu$ |
| SSTORE | Similar to MSTORE, but saves the value onto storage |
| SLOAD | Similar to SLOAD, but takes the value from the storage |
| ADD | Takes 2 (u)int256 from $\mu$, pushes their sum |
| MUL | Takes 2 (u)int256 from $\mu$, pushes their multiplication |
| SUB | Takes 2 (u)int256 from $\mu$, pushes their difference |
| DIV | Takes 2 (u)int256 from $\mu$, pushes their division |

More specifically, the Ethereum bytecode is a sequence of operational codes. It is very similar to other low-level languages, like Assembly, but in Ethereum, to each operational code, is also associated a certain gas cost for performing the operation. Thus, the estimation of non symbolic bounds (i.e. the gas usage for operation different from the `while` command) cannot be done before compiling the code to a simpler formalism, because we need the operational codes for the calculation. This does not apply only to Vyper, but to all high-level Ethereum languages (Solidity, Serpent etc.). Sometimes, it is possible to calculate the gas usage at an intermediary level through abstractions of sets of operational codes, as we will see it is the case for the Vyper compiler.

## 4.2 The Ethereum paradigm

The Ethereum VM is a pre-defined set of rules, used to specify how the Ethereum state ($\sigma$) can change between different blocks. The EVM can be considered as a normal mathematical function, taking the inputs and producing a deterministic output. Thus, we could define the EVM as a state transition function $\Upsilon$.

Ethereum incrementally executes transactions to obtain the current state from an old one. The state includes every data about the blockchain, like addresses, balances and more. The transaction represents a valid arc between two states, where by valid we mean a transaction that follows the implicit rules of the blockchain (e.g., avoid reducing a balance to a negative amount). Formally, their relation can be expressed as:

$$\sigma_{t+1} := \Upsilon(\sigma_t, T)$$

where $\sigma_t$ is the world-state at time $t$, $\Upsilon$ is the state transition function and $T$ is the transition. In other words, $\Upsilon$ is the function that allows changes in Ethereum, while $\sigma$ is where all the current data can be accessed.
The gas estimation is strictly correlated to the $\Upsilon$ function, since its amount depends on the number and type of operations deployed by $\Upsilon$. So, understanding how the state transition function works is needed to define a formula that gives the exact amount of runtime gas used for a certain transaction.
It is assumed that every transaction executed is validated through a set of intrinsic constraints. These include:

- the transaction is well formed, e.g. with no trailing bytes or with a valid nonce;
- the gas limit is no smaller than the intrinsic gas, $g_0$, used by the transaction;
- the sender account balance contains at least the cost, $v_0$ , required in up-front payment.

The intrinsic gas is the gas to be paid before the execution of the transaction. For readability, we define a small fee schedule, that abstracts some common operations deployed during the execution.

| Resized Fee Schedule | | |
|---|---|---|
| Name | Value in gas | Description |
| $G_{txcreate}$ | 32000 | Paid by all contract-creating transactions |
| $G_{txdatazero}$ | 4 | Paid for every zero byte of data or code for a transaction |
| $G_{txdatanonzero}$ | 68 | Paid for every non-zero byte of data or code for a transaction |
| $G_{transaction}$ | 21000 | Paid for every transaction |

**Definition 13.** The intrinsic gas $g_0$ is the gas to be paid before the transaction, obtained as follows:

$$g_0 := \sum_{i \in T_i, T_d} \left\{ \begin{array}{ll} G_{txdatazero} & \text{if } i = 0 \\ G_{txdatanonzero} & \texttt{otherwise} \end{array} \right\} + \left\{ \begin{array}{ll} G_{txcreate} & \text{if } T_t = \emptyset \\ 0 & \texttt{otherwise} \end{array} \right\} + G_{transaction}$$

where $T_i$ is an unlimited size byte array specifying the EVM-code for the account initialization procedure, $T_d$ is an unlimited size byte array specifying the input data of the message, $T_t$ is the target of the transaction, so $T_t = \emptyset$ means there is no target (in other words, this is the account creation transaction).

$T_i$ and $T_d$ are mutual exclusive, this means that $T_i$ exists only if the transaction is for an account creation, while $T_d$ only if it is a message call.

**Definition 14.** Let us call $T_g$ the gas limit set by the end user. Trivially, the computational gas $g_c$ is

$$g_c := T_g - g_0$$

The computational gas is the amount of gas that can be used for the execution, whether it is a message call or a contract creation call. If $g_c$ is not enough to end the execution, then the transaction is reverted and an amount equal to $g_c$ is paid to the miner. We call this amount the runtime gas, $g_r$, and we state that $g_r \leq g_c$ is always verified. This is because, in the other case, the execution finishes before $g_c$ reaches 0, thus the gas paid, $g_r$, must be less.

The runtime gas directly depends on the EVM bytecode. In the previous table we have already seen the abstraction of some operations and their correlated costs. Each of these abstractions is obtained by certain sequences of operational codes. We divide some of the operational codes into subsets as follows:

- $W_{base} = \{\texttt{POP}, ...\}$;
- $W_{verylow} = \{\texttt{ADD},\texttt{SUB},\texttt{PUSH},\texttt{MLOAD},\texttt{MSTORE},, ...\}$;
- $W_{low} = \{\texttt{MUL},\texttt{DIV}, ...\}$;
- $W_{mid} = \{\texttt{JUMP}, ...\}$;

- $W_{high} = \{\texttt{JUMPI}, ...\}$.

Each of these subsets has more operational codes than the ones given in this paper, but as stated in the previous section, we are excluding some codes due to readability reasons.

**Definition 15.** The runtime gas $g_r$ is defined as:

$$g_r := C_{mem}(\mu_f) - C_{mem}(\mu_i) + \left\{ \begin{array}{ll} G_{zero} & \text{if } w \in W_{zero} \\ G_{base} & \text{if } w \in W_{base} \\ G_{verylow} & \text{if } w \in W_{verylow} \\ G_{low} & \text{if } w \in W_{low} \\ G_{mid} & \text{if } w \in W_{mid} \\ G_{high} & \text{if } w \in W_{high} \\ G_{sload} & \text{if } w = \texttt{SLOAD} \\ G_{jumpdest} & \text{if } w = \texttt{JUMPDEST} \\ C_{sstore} & \text{if } w = \texttt{SSTORE} \end{array} \right\} + g_{rest}$$

where $w$ is an operation and

$$C_{mem}(a) \equiv G_{memory} \cdot +a + \frac{a^2}{512}$$

The $C_{mem}$ calculates, based on the number of words, the cost for the extension of the memory, such that all addresses referenced during the computation are valid addresses. On the other hand, $g_{rest}$ is the amount of gas obtained from the excluded operational code. If we also wanted to develop the estimation of $g_{rest}$, we would have needed to add each opcode into the expression between the curly parenthesis. Overall, this is the structure of the formula for the runtime gas.

## 4.3 Compilers

Every compiler follows a general structure, which is divided in a series of phases. Starting from the source code, every phase generates an intermediate representation, which becomes the input of the next phase. Eventually, the last representation is the

object code (i.e. the code written in the target language of the compiler). This subdivision of the compiling process is taken from [15,20,21]. The phases are:

- **Lexical analysis**: it reads the symbols of the source code and groups them into units called *tokens*. As instance, `x = 1`, in Python, returns 3 tokens: the variable `x`, the assign operator `=` and the number `1`;
- **Syntax analysis**: it produces a derivation tree (AST) from a list of tokens. Every leaf of the tree must correspond to a token and, by reading the leaves from left to right, it should be possible to reconstruct a correct sentence of the language. The derivation tree represents the logical structure of the language. Thus, in this phase the compiler also checks that the sequence of tokens is in a certain order, meaning that the program is syntactically well constructed;
- **Semantic analysis**: the tree is subjected to checks relating to the contextual constraints of the language. Contextual constraints could be, as instance, the fact that a variable must be declared before its use, or that the number of actual parameters must be the equal to the number of formal parameters, and more. In this phase, the compiler adds information to each token. For example, a token which is associated to a variable gets information about the type, where it was declared, scope ecc.;
- **Intermediate form**: the code is translated into an intermediate language. It would be possible to obtain, even in this phase, the object code, but it is usually easier to have an intermediate language and, afterwards, perform optimization and generation of the object code;
- **Optimization**: in this phase, the compiler deletes dead code, i.e. code that will never be executed, or performs *inlining*, in which the function calls are substituted with the body of the function that was called. There can also be done optimization on other structures, like over loops or mathematical and boolean expressions;
- **Code generation**: starting from the optimized code, the object code is created.

The first four phases work together with an external structure, called *symbol table*. This table records the symbols, the variables and their associated information, and it is crucial when translating into the intermediate form.

### 4.3.1 Vyper compiler

In general, the Vyper compiler follows this structure, but it has a simpler implementation. Indeed, since Vyper is similar to Python, it is possible to use existing Python libraries to obtain certain code representations, hence avoiding to implement compiler phases. In python, there is already a module, called `ast`, which has a set of built-in functions that can grant syntactical information about a given program. Its function `parse`, given the source code as input, can output the definition tree of the program. Vyper exploits this feature: in the pre-parsing phase, i.e. before obtaining the Abstract Syntax Tree, it reformats vyper source strings into python source ones. Specifically:

- Translates `interface`, `struct` and `event` keywords into python `class` keyword;
- Prevents direct use of python `class` keyword;
- Prevents use of python semi-colon statement separator;

Indeed, `interface`, `struct` and `event` are the only keywords that can be used in Vyper, but not in Python. It is possible to map them into the keyword `class` and let the `ast` module do the parsing. Once the syntax tree has been created, it gets reconverted into a Vyper syntax tree, by recursively evaluating each node and translating python `class` nodes into a Vyper `interface`, `struct` or `event`. This is done by annotating each conversion between Vyper and Python, so that the original values can be obtained from the Python AST. The syntactical checks are done at this point, by checking whether the terms used are part of the language or not. Some notation checking is also performed at this point, e.g. the Vyper language does not allow slicing to occur, so it throws an error if the Python AST has slicing operators. Afterward, constant folding operations are performed on the Vyper syntax tree. Constant folding is the process of statically recognizing constant values and computing them. This avoids calculating constant values at runtime and it can be considered as a form of optimization, since many expressions

will get simplified, hence reducing the complexity of the object code. Then, the compiler extracts information about global structures and data. So, it creates a `globalctx` class, where it stores all the relevant definitions (made inside the global scope). We can say that composite data (structs), Vyper contracts, events, functions and state variables can all be found inside this class. Also, it carves data about imported libraries too. From the contextualized AST, we can get an intermediate representation. The intermediate representation used in Vyper compiler is a LLL[22] (Lisp Like Language). LLL are a family of languages derived from Lisp, which is a functional programming language that has been used in a lot of research areas. It is also often used as an intermediate language during compilation, since its structure and functional nature allow compilers to analyze more easily program properties. Indeed, the gas estimation is calculated on the LLL code, not on the bytecode. This is because it is easier to abstract sequences of operational codes and grouping them into LLL keywords, thus inferring common patterns and gas usage.

**Example 4.3.1.** Consider the following code:

```
@private
def test():
        x: int128 = 0
        y: int128 = 2
        z: int128 = 12 + y
```

Its LLL representation is:

```

# Line 1
[if,
  0,
  [seq,
    /* test() */ [label, priv_4171824493],
    pass,
    /* pop callback pointer */ [mstore, 320, pass],
    pass,
    [mstore, 352, 0],
```

```
    [mstore, 384, 2],
    [mstore,
      416,
      [with,
        l,
        12,
        [with,
          r,
          [mload, 384 <y>],
          [seq, [clamp, [mload, 96], [add, l, r], [mload, 64]]]]]],
    pass,
    [jump, [mload, 320]]]]
```

The LLL representation is close to machine language and it abstracts bytecode operators, like `mstore`, `mload` or `jump`. The keyword `clamp` is used for clamping the first parameter, i.e. restraining its value between the second and third parameter, while `seq` denotes a sequence of commands.

| LLL fee calculation | | |
|---|---|---|
| Name | Value in gas | Description |
| $C_{int}$ | 5 | Gas paid for integer nodes |
| $C_s$ | $gas + 2 * (outs - ins)$ | *Gas* is the cost of the $s$ opcode, *outs* is the stack height at push time, *ins* at pop time |
| $C_{ifelse}$ | $condgas + max(ifgas, elsegas) + 3$ | This is the cost of the condition plus the maximum between the 2 cases |
| $C_{if}$ | $condgas + ifgas + 17$ | This is the cost of the condition plus the cost of the if body |
| $C_{with}$ | $\sum_i (arggas_i) + 5$ | |
| $C_{repeat}$ | $rounds * (bodygas + 50) + 30$ | Gas calculation over iterative looping |
| $C_{seq}$ | $\sum_i (command_i) + 30$ | Gas calculation for consecutive commands |

Table 4.1: LLL fee schedule

The upper bound calculation is done at this stage. The compiler checks each node and assigns an upper bound, based on the type of operation and on the size of memory it

occupies. We can say that, given an LLL representation, the amount of gas it calculates is equal to:

$$gas = \sum_i C_i$$

where $C_i$ is the gas amount for the i-th operation.
The value of $C_i$ depends on the type of operation (which includes `PUSH` and `POP` operations on the stack), based on Table 4.1.

With this calculation, together with the fee schedule for the operational codes, Vyper gets the upper bound over terminating programs. The $C_{repeat}$ cost is the equivalent for the gas calculation over the `for` construct. Indeed, the price depends on the value of *rounds*, which must be known at compile time. Our tool adds another cost, called $C_{while}$, defined as

$$C_{while} = symbound \cdot bodygas$$

where *symbound* is the symbolic formula computed by KoAT over the CITRS.
Afterwards, the intermediate representation gets converted into assembly and, eventually, bytecode, which can be executed on the EVM.

# Chapter 5

# Testing

In this chapter, we provide some examples of algorithms written in Vyper and we calculate the upper bound for their gas usage. We will consider algorithms that use the `while` construct, since otherwise we would get the same output as the original Vyper compiler. To show the correctness of our tool, we will also consider famous algorithms of the complexity theory (thus, with well-known computational effort) and manifest that the computed bound is valid. The coefficients of the polynomials may be overestimated, but that is because, as we explained in Chapter 2, the cost function depends on the number of transitions (steps) inside the CITRS. In other words, instead of only calculating the back-edge transitions, we also consider the transition needed for defining the different branches of the programs, and this increases the coefficients' values.

## 5.1 Fibonacci algorithm

Consider the problem of finding the n-th number of the Fibonacci sequence

0 1 1 2 3 5 8 13 21 34 55 89...

This problem can be solved through recursion, but we can't use it in Vyper. Plus, the algorithm would be computationally expensive. We can write the program as follows:

```
@private
```

```
def fibonacci(limit: int128):
        res: int128 = 0
        l: int128 = limit
        first: int128 = 0
        second: int128 = 1
        tmp : int128 = 0
        if l == 0:
                res = 0
        elif l == 1:
                res = 1
        else:
                l -= 2
                while l > 0:
                        tmp = second
                        second = first + second
                        first = tmp
                        l -= 1
                res = second
        return res
```

The symbolic bound calculation is deployed only for the looping instruction. The CITRS given by our tool for the `while` body is as follows:

```
(GOAL COMPLEXITY)
(STARTTERM (FUNCTIONSYMBOLS l0))
(VAR limit res first second tmp)
(RULES
l0(limit,res,first,second,tmp) -> Com_1(l1(limit,res,first,second,tmp))
l1(limit,res,first,second,tmp) -> Com_1(l2(limit,res,first,second,tmp))
                                     :|: limit > 0
l1(limit,res,first,second,tmp) -> Com_1(l3(limit,res,first,second,tmp))
                                     :|: limit <= 0
l2(limit,res,first,second,tmp) -> Com_1(l1(limit-1,res,second,(first+
                                    second),second)) :|: limit > 0
)
```

Using KoAT, the bound of the while instruction is $4 \cdot \mathit{limit} + 2$ which, as expected, is

linear. By putting together the computed bound with the amount of gas calculated by the compiler, we obtain $493 + 1260 \cdot limit + 630$, which is the maximum amount of gas that the procedure may use.

## 5.2 Insertion sort

Consider an array $a$. Let us try to write a Vyper algorithm that sorts the array using insertion sort.

```
@private
def insertion_sort():
        a: int128[3] = [5,2,10]
        c: int128 = 1
        x: int128 = 3
        while c < 3:
                i: int128 = c-1
                b: int128 = c
                while i>=0:
                        if a[b] < a[i]:
                                tmp: int128 = a[i]
                                a[i] = a[b]
                                a[b] = tmp
                                b -= 1
                        i -= 1
                c += 1
```

A KoAT analysis over the CITRS obtained from this source code gives as output $51c^2 + 142c + 127$. We may as well ignore the coefficients and focus on the fact that the algorithm has a quadratic complexity. Overall, the maximum amount of gas that can be paid is $1062 \cdot (51c^2 + 142c + 127) + 268$

## 5.3 Shell sort

Let us try to write a Vyper algorithm that sorts the array using shell sort.

```
@private
def shellsort():
        lst: int128[9] = [2,5,9,15,44,67,111,423,543]
        n: int128 = 9
        h: int128 = 1
        p: int128 = lst[h]
        while h <= n:
                h = 3*h + 1
        h = h/3
        while h >= 1:
                i: int128 = h+1
                while i < n:
                        tmp: int128 = lst[i]
                        j: int128 = i
                        while j > h and lst[j-h] > tmp:
                                lst[j] = lst[j-h]
                                j = j - h
                        lst[j] = tmp
                        i += 1
                h = h/3
        return
```

A KoAT analysis over the CITRS obtained from the source code gives multiple outputs, depending on the number of non annidated `while` loops. The bound for the first loop is $max([0, -6 + 9 \cdot h]) + max([2, 5 \cdot h])$, while for the second loop is $328 \cdot n \cdot h + 328 \cdot h^2 + 328 \cdot n + 1320 \cdot h + 994$. The first and second loop are $O(n)$ and $O(n^2)$ respectively. Overall, the maximum amount of gas that can be paid is $305 \cdot (max([0, -6 + 9 \cdot h]) + max([2, 5 \cdot h])) + 481 + 635 \cdot (328 \cdot n \cdot h + 328 \cdot h^2 + 328 \cdot n + 1320 \cdot h + 994)$

## 5.4 Binary search

Let us try to write a Vyper algorithm that outputs whether an element is in a given array.

```
@private
def binary_search(item: int128):
```

```
        lst: int128[9] = [2,5,9,15,44,67,111,423,543]
        begin: int128 = 0
        end: int128 = 9
        middle: int128 = 0
        found: int128 = 0
        while begin <= end and found == 0:
                middle = (end + begin)/2
                if lst[middle] == item:
                        found = 1
                else:
                        if item > lst[middle]:
                                begin = middle+1
                        else:
                                end = middle-1
    return
```

In this case, KoAT is not able to find an upper bound, thus it outputs $\infty$. This is probably because complexity analysis tools usually have difficulties in creating an appropriate ranking function for logarithmic algorithms.

# Conclusions

We have obtained a new Vyper's compiler that allows infinite looping. We have devised a way to translate programs written in Vyper into CITRS and, afterward, calculate a symbolic upper bound for the complexity of the system. We have seen that the tool may require too much computational effort when arrays are used. Unfortunately, at compile time it is not possible to understand which element of the array is being accessed. So, the tool has to consider and create a rewriting rule for every possible element, otherwise it may output an incomplete CITRS, which may not behave like the original program in all the cases. We have noticed that the tool may give $\infty$ bounds in case of logarithmic computations. This is due to the limits of complexity analyzers. Indeed, the literature[23,24], it is known that they struggle to find precise upper bounds for these kind of programs. Future developments should try to simplify the `(ListAssign)` reduction rule. There are some algorithms which can reduce the domain of elements that may be accessed for a specific command. By implementing this algorithm[25], we could obtain a simplified CITRS and a better performing Vyper compiler. This new Vyper compiler cannot replace the original one. Indeed, the original compiler, as stated in Chapter 1, provides more safety properties and can output a precise upper bound. This extended compiler, instead, may fail to give upper bounds or, in general, is less precise and less secure. On the other hand, our language is more expressive and it is more similar to Solidity. Solidity is a javascript-based language for Ethereum and it is the most used one for dApps. Future researches could optimize the compiler presented in this thesis and create a more stable and usable turing-complete python-based language for Ethereum.

# Bibliography

[1] J.W.Klop (1992). *Term Rewriting Systems, Handbook of Logic in Computer Science.* Oxford University Press.

[2] N.Dershowitz & M.Okada (1988). *Proof-Theoretic Techniques for Term Rewrite Theory.* Proc. 3rd IEEE Symp. on Logic in Computer Science.

[3] D.Dougherty, P.Lescanne, L.Liquori & F.Lang (2005). *Addressed Term Rewriting Systems: Syntax, Semantics, and Pragmatics.* Electronic Notes in Theoretical Computer Science 127, 57–82.

[4] G.Wood (2013). *Ethereum: a Secure Decentralised Generalised Transaction Ledger* [https://ethereum.github.io/yellowpaper/paper.pdf].

[5] B.Hauser (2015). *Vyper documentation* [https://vyper.readthedocs.io/en/]

[6] M.Di Pirro, S.Crafa and E.Zucca (2020). *Is Solidity Solid Enough?*. International Conference on Financial Cryptography and Data Security. Financial Cryptography Workshops, pp 138-153.

[7] M.Di Pirro (2018). *How Solid is Solidity? An In-Depth Study of Solidity's Type Safety.* Technical report. Università degli Studi di Padova.

[8] A.Mavridou & A.Laszka (2018). *Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach.* International Conference on Financial Cryptography and Data Security. Financial Cryptography and Data Security, pp 523-540.

[9] A.Mavridou & A.Laszka (2017). *Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach.* Technical report. Cornell University.

[10] P.Merriam (2020). *Stateless Clients: A New Direction for Ethereum 1.x.* Medium.

[11] R.Sierra (2019). *Verification of Ethereum Smart Contracts Written in Vyper* [https://www.pm.inf.ethz.ch/education/student-projects/completedprojects.html]. University of Zurich.

[12] M.Ghelli (2020). *Analisi Statica di Smart Contract in Ethereum: Stimare i Consumi di Gas* [https://amslaurea.unibo.it/19612/]. Alma Mater Studiorum - Università di Bologna.

[13] R.N.Moll, M.A.Arbib and A.J. Kloury (1988). *An Introduction to Formal language Theory.* Springer Verlag.

[14] E.Hildenbrandt, M.Saxena, X.Zhu, N.Rodrigues, P.Da- ian, D.Guth and G.Rosu (2017). *Kevm: A complete semantics of the ethereum virtual machine.*

[15] M.Gabbrielli and S.Martini (2011). *Programming Languages: Principles and Paradigms.* McGraw-Hill.

[16] A.Bertossi and A.Montresor (2014). *Algoritmi e Strutture di Dati.* CittàStudi.

[17] S.Nakamoto (2008). *Bitcoin: A peer-to-peer electronic cash system*[https://www.bitcoin.com/get-started/bitcoin-white-paper-beginner-guide/].

[18] N.Dershowitz and Z.Manna (1979). *Proving termination with multiset orderings.* Commun. ACM 22, 465-476.

[19] M. Martel (2008). *Semantics Transformation of Arithmetic Expressions.* Université Paris-Saclay.

[20] A.V. Aho, M.S.Lam, R.Sethi and J.D.Ullman (2007). *Compilers: Principles, Techniques and Tools.* Second Edition. Pearson.

[21] A.W.Appel (2007). *Modern Compiler Implementation in C.* University of Cambridge.

[22] J.McCarthy (1985). *LISP 1.5 Programmer's Manual.* MIT Press.

[23] S.Falke, R.Thiemann, J.Giesl and P.Schneider-Kamp (2003). *AProVE: A System for Proving Termination*. University of Aachen.

[24] M.Brockschmidt, F.Emmes, S.Falke, C.Fuhs, J.Giesl (2003). *Analyzing Runtime and Size Complexity of Integer Programs*. ACM Transactions on Programming Languages and Systems.

[25] Y.Paek, J.Hoeflinger, D.Padua (2002).*Efficient and precise array access analysis*. ACM Transactions on Programming Languages and Systems.

[26] T.H.Cormen, C.E.Leiserson, R.L.Rivest and C.Stein (1990). *Introduction to Algorithms*. MIT Press.

# Ringraziamenti

Ringrazio la mia famiglia per avermi supportato durante questo percorso. Ringrazio il professor Dal Lago e il professor Vanoni per il loro impegno e per avermi aiutato nella stesura di questa tesi. Ringrazio l'intera università per il livello di istruzione che mi ha dato.