Formal analysis and gas estimation for Ethereum smart contracts

Beatriz Henriques Xavier

Thesis to obtain the Master of Science Degree in

Mathematics and Applications

Supervisor: Prof. Paulo Alexandre Carreira Mateus

Examination Committee

Chairperson: Prof. Maria Cristina de Sales Viana Serôdio Sernadas

Supervisor: Prof. Paulo Alexandre Carreira Mateus

Member of the Committee: Prof. Pedro Miguel dos Santos Alves Madeira Adão

December 2018

Resumo

A plataforma Ethereum constitui um ecossistema para o desenvolvimento de *smart contracts* – programas que se encontram guardados numa *blockchain* pública e que podem interagir entre si. Sistemas de verificação formal para estas plataformas são muito importantes, uma vez que as modificações feitas na *blockchain* são irreversíveis e podem envolver valores monetários.

Nesta dissertação estudamos a máquina virtual de Ethereum e implementamos uma máquina virtual de Ethereum simbólica em Mathematica na qual é possível executar *smart contracts* com *input* simbólico. Esta máquina é utilizada no desenvolvimento de uma ferramenta que simula todos os caminhos possíveis num *smart contract* e que, no final, devolve todos os estados finais possíveis do sistema. Esta abordagem exaustiva permite a verificação de propriedades acerca do estado global do sistema após a execução de um contrato. Além disso, estendemos as regras semânticas da máquina clássica à máquina simbólica.

A execução de código na *blockchain* tem um custo que depende do estado global do sistema e do *input* dado ao código. Exploramos a temática dos custos em *gas* e utilizamos a máquina virtual simbólica na construção de um programa que estima o custo da execução de uma função de um contrato. Esta vai além do que era alcançado por trabalhos anteriores dado que não só considera uma classe de contratos mais alargada como também devolve informação mais detalhada, nomeadamente, aceita código que contém ciclos com um número fixo de iterações e devolve o custo de cada caminho de execução em vez de um único majorante.

Palavras-chave: verificação formal, execução simbólica, análise de controlo de fluxo, semântica operacional, Ethereum, blockchain

Abstract

Ethereum provides an ecosystem for the development of smart contracts – programs stored in a public blockchain that may interact with each other. Formal verification systems for these platforms are of great importance since the modifications made on the blockchain are irreversible and may concern tokens with monetary value.

We study the Ethereum Virtual Machine and implement a symbolic Ethereum Virtual Machine in Mathematica that is able to run smart contracts with symbolic input parameters. This machine is used in the development of a formal analysis tool that symbolically executes every execution path of a smart contract, returning all possible final states of the system. This exhaustive approach allows a user to verify properties about the global state of the system after the execution of a given contract. Moreover, we extend the semantics of the classic virtual machine to this symbolic virtual machine.

Executing code on the blockchain has a cost that depends both on the global state of the system and on the input given to the code. We explore the topic of gas costs and use the symbolic virtual machine to build a tool that manages to estimate the cost of executing a function of a contract that extends previously existing tools since it not only considers a broader class of contracts, but also returns more detailed information. Namely, the tool accepts code containing loops with a fixed number of iterations and returns a cost for every execution path, rather than a single upper estimate.

Keywords: formal verification, symbolic execution, control flow analysis, operational semantics, Ethereum, blockchain

Contents

List of Figures

Chapter 1

Introduction

1.1 Background and motivation

Ethereum is a platform designed to serve as a ledger of financial transactions and decentralized applications. Inspired by the Bitcoin, launched in 2009 [13], Vitalik Buterin created and first described Ethereum in early 2014 in a white paper [2]. Bitcoin is a cryptocurrency system functioning as a bank in a public, decentralized database called blockchain. Transactions are validated by some users, the miners, through cryptographic algorithms that verify their legitimacy – a process known as proof of work – and stored in blocks. The blockchain is an ordered sequence of blocks. Thus, the blockchain is built collectively by the users and the accepted state of the blockchain at any moment is determined by the majority of the miners. This consensus mechanism eliminates the need for a central authority and distributes the power and the responsibility equitatively by the users. Bitcoin, as other digital currencies, has no intrinsic value as it is not issued by any central entity; the value is attributed to the coin by those who use it.

The main idea behind the development of Ethereum was to expand the concept of blockchain present in Bitcoin to other applications besides monetary transactions, such as proving property ownership, establishing decentralized autonomous organizations, voting, gambling, or attributing automatic insurance compensations. Wanting to be able to encompass essentially everything that can be programmed, Ethereum developed a virtual machine that specifies a Turing-complete language over which these applications, called smart contracts, must be written. There is, however, a limitation on the amount of computation that a program can trigger. Ethereum introduced the concept of gas: every operation has a cost in gas and every program execution has an upper limit on the amount of gas that can be spent. The network has its own currency, ether, given as a reward to the miners that contribute to the maintenance of the blockchain. The full description of the Ethereum environment and of the Ethereum Virtual Machine (EVM) was published in late 2014 in a yellow paper by the co-founder Gavin Wood [19]. It specifies the structure of the blockchain, transactions and accounts; the execution model for EVM code; the proof of work used by the system to avoid double spending; and the regulatory scheme regarding fees and rewards, which is very closely related to gas.

Smart contracts were originally designed by Nick Szabo in 1994 [16] to be programs that implement

real contracts, whether business or personal, specifying the conditions on which the involved parties agree and the consequences of the commitment. This idea became more powerful considering the public and definitive character of the blockchain, and smart contracts evolved to be any functionality provided by a computer program. Contracts are intended to be a part of the global Ethereum ecosystem as they are public and they can interact with each other. A smart contract can contain instructions to call any smart contract; the outcome is determined by the contents of the called contract.

Given that Ethereum wishes to be a project as open and global as possible, it also contains high-level languages that are compiled to EVM bytecode. The most popular one is Solidity, an object-oriented programming language similar to JavaScript. Smart contracts are then similar to classes that contain attributes and functions. Several implementations of the Ethereum Virtual Machine and of the blockchain, called Ethereum clients, are freely available – cpp-ethereum is based on C++, go-ethereum (also known as geth) on Go, and pyethapp on Python, just to name a few.

The first phase of the Ethereum blockchain, Frontier, as well as a preliminary version of Solidity and some Ethereum clients started in July 2015 with the release of the first block, named genesis, over which the blockchain was intended to be built. As of September 2018, there are around 6 million blocks in the accepted version of the blockchain, 300 million transactions were committed, and over 40 million accounts and smart contracts were deployed.

New digital currencies have been developed by influence of Ethereum or over Ethereum, since that is one of the applications that is possible to program in a smart contract. Other applications were already developed by individuals or companies. The insurance company AXA released in 2018 a service to automatically provide compensation for flight delays, allowing the client to choose a coverage plan with clear conditions and consequences, using flight information provided by independent parties, and promoting transparency by storing the smart contract in the blockchain. Acronis launched in 2016 a decentralized notary service that proves the ownership, contents and date of signature of an encrypted file; the consensus mechanism turns difficult to forge a document. A casino named Etheroll ensures that the bets are fair by generating random numbers using hash functions and that the house wins are small, which can be verified by everyone since every game is recorded in the blockchain.

1.2 Goal and achievements

One major security concern arises precisely from the fact that it is really easy to program a smart contract: Solidity resembles most programming languages and contracts are typically small, modularized pieces of code; but a minor error in the implementation of the EVM or a misunderstanding of a developer about the features of the language can cause a user to lose real money. Our work focuses on the formal analysis of Ethereum bytecode.

Although smart contracts are usually written in diverse high-level languages, every piece of code is compiled to the common core of the Ethereum network that is EVM bytecode. For that reason, we have decided that a complete analysis of smart contracts should be performed at the bytecode level. In a framework as plural as Ethereum, that combines currency and functionality, and that was designed to be

used at a global scale, a formal verification system that provides the developers a better understanding about the behaviour of their contracts is of the uttermost importance.

The short history of Ethereum already contains some attacks to smart contracts that caused their owners to lose ether. It is worth to mention the exploits to a contract named The Decentralized Autonomous Organization (The DAO) in 2016, an automated capital venture fund from which 3.6 million ether (around 50 million dollars at the time) were stolen and, more recently, in 2018, to the PoWH (Proof of Weak Hands) Coin contract, from which 866 ether (around 800 thousand dollars at the time) were inappropriately taken. The Ethereum community has identified some known types of vulnerabilities and has issued recommendations for the developers in order to avoid these particular programming flaws, but the need for a complete formal analysis of smart contracts persists.

In this dissertation, developed in collaboration with Tekever, we have designed and implemented in Mathematica an abstraction of the Ethereum Virtual Machine that is able to symbolically execute interactions between given smart contracts and return all possible final states of the system – resultant from all possible execution paths – and the conditions under each path is taken. This outcome allows us to verify properties about the executions by asking the question: is there a final state that violates the desired property? If the answer is negative, then the contract is safe relatively to that property and to the interactions with the other specified contracts; if the answer is positive, we are able to traceback the paths that lead to the final states that do not respect the property and find out the portion of the code that is behaving unexpectedly. As an example of property, one might be interested in knowing if its balance can decrease by calling a given function of other contract.

We specify the semantics of this symbolic machine, a generalization of the semantics developed for the Ethereum Virtual Machine by Grishchenko et al. [6], highlighting the differences between the two systems. These semantics reflect the main features and limitations of the symbolic model.

The symbolic machine also provided us the necessary tools to build a gas estimator for smart contracts. This estimator is able to simulate all execution paths of a contract and return a list of possible costs along with the conditions under which each cost holds. It is able to process code without loops and with loops with a fixed number of iterations, which constitutes an improvement over previously existing tools. The analysis of loops performed by this tool was based on the study of structured loops and of an algorithm that determines if all loops of a program are natural or not.

To summarize, the main contributions of this work are the following:

- Implementation of a symbolic Ethereum Virtual Machine in Mathematica. It is possible to execute a function of a smart contract with symbolic input and considering symbolic account contents and to retrieve a comprehensive list of the possible final states;
- Adaptation of the small-step semantic rules of the classic virtual machine to the symbolic virtual machine;
- Development of a gas estimator for smart contracts that contain loops with a fixed number of iterations;

- Adaptation of Tarjan's algorithm to return all natural loops of a program more efficiently. We have also detailed the proofs of some results that were stated without proof in the original paper [17] and in the book on algorithms [4] about the correction and computational complexity of Tarjan's algorithm.

1.3 Literature review

The main bibliographical sources for the development of the formal system were the original specification of the EVM by Wood [19] and the semantics of the EVM formalized by Grishchenko et al. [6]. We were also inspired by an early version of the semantics of the EVM published by Luu et al. in 2016 [12]. Related work on formalization of the EVM includes the implementation of the EVM in Lem – a language that can be compiled to theorem provers like Isabelle/HOL – by Yoichi Hirai from the Ethereum Foundation [11], and in the $\mathbb{K}$ framework – a language designed specifically for rewriting semantics systems – by Hildenbrandt et al. [10]. In what concerns gas estimation and optimization, although headed in a different direction, it is worth mentioning the tool Gasper [3], developed in 2016, that automatically identifies predefined costly programming patterns and replaces them by less expensive pieces of code.

Several theoretical results about control flow graphs of programs and loop structuring by Tarjan [17] and Hecht and Ullman [9] were essential to build a mathematical basis for the gas cost estimation and bytecode analysis chapter. The complexity results that were achieved were based on the classic book on algorithms by Cormen et al. [4].

Finally, the available online information and repositories maintained by the Ethereum community played a very important role throughout the entire work.

1.4 Outline

Chapter 2 introduces the Ethereum framework, defining the components that make up the Ethereum environment. The virtual machine architecture and operations are described, and the relation between the machine and the environment is detailed. Moreover, the semantics of the operations are briefly explained.

In Chapter 3 we examine the symbolic model. It starts with a general description of the functionalities and motivation for some needed design decisions and continues by formally defining what is meant by condition of an execution path. Later, the semantic rules that govern the developed model are specified. The chapter finishes with the analysis of a real smart contract that was exploited in 2018. We define a property that should be true after every execution – according to what we believe that was the intended behaviour of the program – and find an execution path that violates that property.

Chapter 4 explores a different subject: it concerns estimation of gas costs of executing smart contracts. Program structure analysis, with primary focus on loops, is studied and an adaptation of an algorithm by Tarjan that efficiently finds all natural loops of a program from its control flow graph is developed. A tool that estimates the cost of executing a function of a contract is developed and explored through real

examples.

The conclusions of this work are summarized and future research directions are suggested in Chapter 5.

Chapter 2

The Ethereum Virtual Machine

This chapter introduces the Ethereum blockchain and the Ethereum Virtual Machine in more detail. It was already mentioned in the introduction that the blockchain is a public ledger that contains information about the global state and history of the system. This includes transaction details, such as senders, recipients, values, and dates, as well as account contents. Accounts are a central concept of Ethereum. An account can store, send, and receive ether, similarly to a bank account. However, Ethereum accounts have more functionalities: internal storage and executable code. A *smart contract* is simply an account that has associated code, as we will see in the following section.

A (process) virtual machine is an abstraction that allows an application to run independently of the underlying hardware. The Ethereum Virtual Machine provides a programming language (a finite set of operations) and an environment for the programs to run constituted by elements such as a volatile memory and an execution stack. The specification of the EVM can be found in the original description of Ethereum by Gavin Wood, [19]. This controlled environment ensures that Ethereum programs are independent from previously existing frameworks and their possible flaws. The semantics of the EVM were recently formalized by Grishchenko et al. in [6]. We will give an overview of the EVM operations, highlighting its most important and distinctive features relatively to other systems, and introduce the definitions of some concepts related to the execution of smart contracts that are needed to develop the semantics of the operations and to perform formal analysis of programs.

The EVM is a stack-based machine working on arithmetic modulo 2^{256}. Its elements are, thus, 256-bit numbers. Its operations, also known as opcodes, form a Turing-complete language, but the machine has an additional feature: the amount of computational work done in every execution is bounded by *gas*. Gas has a cost in ether and the gas given to the execution of a transaction is specified by the user that started it. Every operation has a positive gas cost. If the execution runs out of gas, it stops and the changes that it made to the system are reverted. This measure guarantees that every execution eventually stops and places an upper limit on the fee that the sender must pay to deploy a transaction to the blockchain. This is particularly important in a system where all programs may interact with each other and every nested execution is paid by the original sender. Gas is the reason why the EVM is commonly referred to as a *quasi*-Turing complete virtual machine.

Ethereum uses the standard cryptographic hash functions `KECCAK-256` and `KECCAK-512` for the generation of account addresses, signatures of transactions, validation of blocks, among others. In the present work, the expression *hash function* should be read as *cryptographic hash function*. Although everything on the Ethereum blockchain – account, transaction, and block contents – is public, some information may be encrypted using these functions, which means that one can prove his identity, or the value of any blockchain component that he knows, without revealing it.

The following section establishes the formal definitions of the basic concepts of Ethereum.

2.1 Basic definitions

The formal definitions presented herein are a simplification of the original ones given in [19] and formalized in [6] and only contain the elements that are necessary to our analysis. Let $\mathbb{A} = \{0,1\}^{160}$ be the set of all possible addresses, $\mathbb{B} = \{0,\dots,255\}$ the set of bytes and $\mathbb{N}_{256}$ the set of natural numbers below 2^{256}. S^* denotes the Kleene closure of the set S and ε denotes the symbol such that $\{\varepsilon\}$ is the empty element of S^*.

The formal definition of *account* is relative to account contents; the link between addresses and account contents is a part of the global state of the system.

Definition 1 (Account)**.** *An account is a tuple $A = (b, code, stor, n)$ where:*

- $b \in \mathbb{N}_{256}$ *is the balance;*
- $code \in \mathbb{B}^*$ *is the code. The account is a contract if $code \neq \varepsilon$ and externally owned otherwise;*
- *stor is a function from $\{0,1\}^{256}$ to $\{0,1\}^{256}$ and it is the storage;*
- $n \in \mathbb{N}_{256}$ *is the nonce. If the account is externally owned, n is the number of transactions sent from it. If the account is a contract, n is the number of account creations made by the account.*

The nonce of an account is simply a counter of transactions or contract creations. This should not be confused with the nonce of a block, which will be introduced later. The storage of an account is a permanent repository whose contents are stored in a cryptographically secure and efficient data structure called Merkle Patricia tree. The code of a contract can be called by any other contract. Externally owned accounts are accessed using private keys and contracts are only interactable with through execution of code. Accounts are identified by their addresses and the state of every account in the Ethereum network constitutes the global state of the system.

Definition 2 (Global state)**.** *The global state of the system is a function between the set of addresses and the set of accounts:*

$$\sigma \colon \mathbb{A} \to \mathbb{N}_{256} \times \mathbb{B}^* \times (\{0,1\}^{256} \to \{0,1\}^{256}) \times \mathbb{N}_{256}$$

If $\sigma(a) = (b, code, stor, n)$ we say that a is the *address* of the account $(b, code, stor, n)$. We will abuse the notation by using the word *account* to mean the entity constituted by the address a and the account

$\sigma(a)$. σ should map elements $a \in \mathbb{A}$ to the balance, code, storage and nonce of the account with address a in the Ethereum blockchain. Obviously, the value of $\sigma(a)$ depends on the time: balances, storage contents and nonces of accounts change with transferences and execution of code. Rigorously, the function σ should also depend on a time parameter, but we do not write it explicitly in order to simplify the notation. *code* is the only immutable field of an account and it is defined during its creation. We convention that the account with address a *exists* if $\sigma(a) \neq (0, \varepsilon, 0_F, 0)$, where 0_F is the constant function whose value is always zero.

Transactions are the means of communication between accounts. A transaction contains transference of ether and also instructions to trigger the execution of the code of the recipient.

Definition 3 (Transaction)**.** *A transaction is a tuple* $T = (from, to, n, v, d, g, p)$*, where:*

- $from, to \in \mathbb{A}$ *are the sender and the receiver addresses of the transaction, respectively;*
- $n \in \mathbb{N}_{256}$ *is the nonce of the account with address* $from$*;*
- $v \in \mathbb{N}_{256}$ *is the value;*
- $d \in \mathbb{B}^*$ *is the data;*
- $g \in \mathbb{N}_{256}$ *is the gas limit: the maximum amount of gas that should be used in this transaction;*
- $p \in \mathbb{N}_{256}$ *is the gas price.*

If $to = 0$*, then* T *is an account creation transaction and* d *is code used to determine the code of the new account. Otherwise,* d *is the data given as input to the execution of the code of the account with address* to*.*

We will denote the set of transaction contents by $\mathbb{T} = \mathbb{A} \times \mathbb{A} \times \mathbb{N}_{256} \times \mathbb{N}_{256} \times \mathbb{B}^* \times \mathbb{N}_{256} \times \mathbb{N}_{256}$. An account creation transaction results in the deployment of a new account A to the blockchain, which updates the global state function σ. The address of A is the number formed by the last 160 bits of the result of the hash function `KECCAK-256` applied to an encoding of $from$ and n. The probability of overwriting an already existing address is, thus, extremely small. The balance of A is v, the value transferred in the transaction. The code of A is the output of the execution of d. The storage of A is the zero function and the nonce of A is 0.

A transaction where $to \neq 0$ results in the increase of the balance of the account to by v units, in the decrease of the balance of the account $from$ by $v + gp$ units, and in every change to the global state made by the execution of the code of the account to with input d.

When a transaction is broadcasted to the blockchain, it will be executed by some users, called *miners*. Since executing a transaction means executing code, every transaction involves the payment of a fee in ether, equal to the gas spent times the gas price. The gas price is chosen by the sender and the gas spent is never greater than the gas limit, so the sender knows upfront the maximum fee that he may have to pay. Indeed, if his balance is less than this maximum fee, the transaction is declared invalid. If the execution runs out of gas before finishing, the sender pays the maximum fee but the transaction is

not completed and every change that was made to the system during the partial execution is reverted. If there is enough gas, the transaction is completed and the miners start trying to solve the problem that constitutes the proof of work. The first miner that completes the proof of work broadcasts that information and the other miners check that he did it. Then every miner executes the transaction and updates its own database to reflect the changes that were made to the system.

Gas is a concept that only exists inside a transaction. The detachment between the notions of gas and ether exists so that the fees do not depend linearly on the current value of ether. Gas costs of operations are predetermined and intend to reflect the computational effort that they require. Costs are either constant or dependent on the operands, but do not change between transactions. Gas prices, on the other hand, are set by users who want to deploy a transaction. They are not chosen depending only on the value of ether, but on the priority that the user implicitly attributes to the transaction: miners can choose which transactions they will execute and tend to prefer the ones with higher gas prices; the transaction sender is interested in keeping the gas price low, as long as there is a miner interested in validating his transaction.

Double spending is a problem that is common to every digital currency. Since money is not physical, it is necessary to guarantee that a user cannot spend more than the amount that he has in his account, or "spend the same coin twice". Ethereum deals with this situation using the nonce n of an account in a transaction. A dishonest account A with balance x could try to transfer y to an account B and z to an account C, such that $y \leq x, z \leq x$ and $y + z > x$. If the transaction to C is broadcasted before the transaction to B is accepted, the balance of A is x when the transaction to C is broadcasted, so it is not rejected due to lack of funds. If the transaction to C is sent with a higher gas price, it is likely that miners will try to validate it first. But the nonce of the second transaction is not equal to one plus the last nonce used by A in a transaction, so the transaction is determined to be invalid.

Transactions are recorded inside blocks, the fundamental unit of the blockchain. Blocks contain several components, including references to previous blocks and to data structures that store information about the transactions of the block; the following definition is a simplification.

Definition 4 (Block)**.** *A block is a tuple* $B = (nonce, \mathcal{T}, i, l, ben, d)$ *where:*

- $nonce \in \mathbb{B}^8$ *is the nonce.*
- $\mathcal{T} = (T_1, \ldots, T_m)$ *is a list of transactions.*
- $i \in \mathbb{N}$ *is the number.*
- $l \in \mathbb{N}$ *is the gas limit.*
- $ben \in \mathbb{A}$ *is the address of the account of the beneficiary;*
- $d \in \mathbb{N}$ *is the difficulty.*

We will denote the set of block contents by $\mathcal{B}$. Naturally, $\mathcal{B} = \mathbb{B}_8 \times \mathbb{T} \times \mathbb{N} \times \mathbb{N} \times \mathbb{A} \times \mathbb{N}$, where $\mathbb{T}$ is the set of transaction contents. Since the aim of the blockchain is to be a decentralized platform, every user has its own version of it. However, it is the consensus between the users that allows such a system

to work. We convention that the word *blockchain* refers to the most commonly accepted version of the Ethereum blockchain.

The first block ever deployed to the blockchain, named *genesis*, has number $i = 0$. The number of the successor of the block number i, B_i, is the block B_{i+1}. The term *successor* is relative to the order of acceptance of the blocks in the blockchain. The nonce of a block is a number that can be written with 8 bytes, or 64 bits, and it is used in the verification of its validity.

Definition 5 (Blockchain)**.** *The blockchain is an ordered sequence of blocks,* $(B_0, B_1, \ldots, B_k)$*, such that* $B_0 = (42, \emptyset, 0, 3141592, 0, 2^{17})$.

The genesis block B_0 is fixed; all further evolution of the blockchain depends on the miners. Every block in the blockchain must be valid. The validity of a block is checked through several procedures, including checking the validity of every transaction (confirming the nonce and verifying the signature of the sender) and the proof of work. This concept was introduced in 1992 by Cynthia Dwork and Moni Naor in [5] with the intent of disencouraging spam emails. The general idea of a proof of work system is to use a function f such that, for any x, computing the value of $f(x)$ requires a lot of computational resources but, given x and y, checking if $y = f(x)$ can be done very quickly by a computer. Proof-of-work systems are used in cryptocurrencies in order to give value to the currency: they show that a reasonable amount of work was done during their creation.

2.2 Ethereum proof of work

Blocks store information about every transaction that was ever made. Therefore, when a block is sent to the blockchain, every user must agree that that block will become a part of blockchain. In order to achieve this consensus, the miners actively participate in the process of validating blocks in exchange for rewards when they are successful.

When a block is sent to the blockchain it is incomplete: it does not have a nonce yet. Validating a block consists of finding a nonce for the block that satisfies some conditions. The proof of work done by a miner whose account address is a for an incomplete block $\tilde{B} = (\mathcal{T}, i, l, a, d)$ can be roughly described in a few steps:

1. Compute $target = \dfrac{2^{256}}{d}$;
2. Construct a big data set D that depends on i. The steps needed to compute D involve hashing i and some values used in the intermediate calculations of the data set of previous blocks;
3. Guess an integer x between 0 and $2^{64} - 1$;
4. Compute $\texttt{PoW}(\tilde{B}, x, D)$. `PoW` is a function that involves encoding and hashing the components of $\tilde{B}$, the tentative nonce x and some elements of D, and that returns a pair (m, y);
5. If $y > target$, set x to $x + 1 \mod 2^{64}$ and go to step 4. Otherwise, set x as the nonce of $\tilde{B}$. The final block is $B = (x, \mathcal{T}, i, l, a, d)$.

Every miner runs this algorithm. When a miner finds a valid nonce n, he broadcasts it to the network and every other miner can compute $\texttt{PoW}(\tilde{B}, n, D)$, using the address of the successful miner as a beneficiary of $\tilde{B}$, and check that it has the desired property. Being based on hash functions, `PoW` can be computed quickly, but it is hard to find an input such that the output fulfills some condition. Thus, a miner that finds a valid nonce proved that he has done some work in the maintenance of the blockchain, hence the name proof of work, and receives a reward in ether.

The proof of work helps to solve the problem of double spending: if the malicious account A with balance x tries to transfer y to B and z to C as before, and one of the transactions is accepted, when the miners are validating the other transaction they will find out that A has used all its balance in a previous transaction, an information that they can all confirm by computing `PoW` using the nonce of the block of the first transaction. Then, the malicious transaction will be rejected. Account A would have to be able to rewrite the blockchain history and have the other miners accept it in order to claim that he never spent that ether, but that would take too much time due to the proof of work algorithm. The average time that it takes to find a valid nonce depends on the difficulty of the block, so this value is frequently adjusted in order to control the time between blocks.

This process also validates the identity of the miner that first found the nonce, since changing the beneficiary address would also change the output of the `PoW` function considering the same nonce n, which would be greater than *target* with a very high probability. Therefore, it is hard for a malicious user to steal a valid nonce from a miner who performed work to find it and claim it as his.

2.3 Ethereum Virtual Machine overview

The architecture of the EVM is simple: it is a stack machine that operates on 256-bit elements. The original specification of the stack machine contains 129 opcodes with diverse functionalities, from simple arithmetic operations to account creation and deletion. The operands of each instruction, when they exist, are popped from the top of the stack, and the result is pushed to the stack. The stack has capacity for 1024 elements. If it overflows, an exception is raised and all changes made during the execution are lost. An opcode is represented by a number from 0 to 255. A program is a finite sequence of bytes – if a byte does not represent any existing opcode, we say that it is an *invalid* opcode.

An execution of a program has a volatile memory associated with it, defined as an infinite bytearray. In the beginning of the execution, every position of the memory contains the number 0. The virtual machine contains an operation, `MSTORE`, that writes an element of the stack in a specified position the memory. The difference between the designs of the stack and of the memory causes the writing operation to split the element to store in 32 bytes and to write each byte in a contiguous position of the memory, starting from the specified position. It is also possible to access the memory and copy an element formed by 32 contiguous bytes of it to the stack, using the operation `MLOAD`. This memory is cleared when the execution finishes but has the advantage of being much cheaper (in terms of gas) than a permanent storage, so it is typically used to store values of local variables that are only used to determine the output of an execution of a program.

The machine is also able to communicate with the internal storage of the account whose code is executing. This storage can be accessed and modified by the virtual machine – opcodes `SLOAD` and `SSTORE`, respectively – but, unlike what happens with the memory, the changes made to the storage last after the execution finishes. Hence, the storage is not cleared when an execution starts; it contains what was stored there in previous executions of the program. Although we formally defined storage as a function, its natural structure is a set of key-value pairs, and that is indeed followed by most implementations of the EVM. It is normally used to write the values of global variables of a program: for instance, if a smart contract implements a bank, the amount that each client owns may be updated by functions such as *withdraw* or *deposit* and must not be lost between executions.

Since the word size of the EVM is 256 bits, all arithmetic operations are modulo 2^{256}. A few examples are `ADD`, `MUL`, `SUB`, `DIV`, `MOD`, `EXP`. Their semantics are as expected. Other arithmetic instructions interpret their operands as two's complement signed 256-bit integers. There are several comparison operators that return 1 or 0 if the comparison returns true or false, respectively: `LT`, `GT`, `EQ`, bitwise `AND`, bitwise `OR`, bitwise `XOR` and bitwise `NOT` are some of them. It is also possible to push an element with n bytes to the stack (`PUSH`n, $1 \leq n \leq 32$), to `POP` the top element, to duplicate an existing element up to the 16th position of the stack (`DUP`n, $1 \leq n \leq 16$), or to swap the first element with other member of the stack, up to the 17th (`SWAP`n, $1 \leq n \leq 16$). Finishing an execution with or without output corresponds to executing `RETURN` or `STOP`.

The cryptographic hash function `KECCAK-256` is embedded in the virtual machine through the opcode `SHA3`. `KECCAK-256` is a hash function from the Keccak family that, in 2012, won the public contest promoted by the National Institute of Standards and Technology (NIST) for the development of SHA-3. After the Ethereum release, NIST made a minor change in the algorithm concerning the padding of the input. Therefore, the hash function used by Ethereum is not the current standard SHA-3, but the security provided by both functions is equivalent.

The virtual machine provides access to the environment where the program is executed: some opcodes concern block information – `NUMBER`, `DIFFICULTY`, `GASLIMIT` –, some are relative to the transaction – `CALLER` for the address of the sender, `CALLVALUE` for the value, `CALLDATALOAD` for the data –, and others access information about other accounts – `BALANCE` to get the balance of an account, `EXTCODECOPY` to get its code.

The most distinctive and important operations of the EVM are related to creation, interaction and destruction of accounts. Account creation is achieved using `CREATE`. This operation generates a new address, transfers the specified value to the new account, and causes the current execution to pause, starting the execution of the code d that contains instructions to build the code of the new account in a fresh execution stack with a fresh memory. When this execution finishes, the original one continues. It is also possible to destroy an account through the opcode `SELFDESTRUCT` (named `SUICIDE` in the original specification). It receives as an operand the address of the account that should receive the balance of the deleted account. Finally, a transaction from an account A to an account B may be performed by three opcodes: `CALL` and its variants `CALLCODE` and `DELEGATECALL`. The first action is the transference of the designated value to B. If B is an externally owned account, the transaction finishes. Otherwise, similarly

to what happens during account creation, the code of B runs with the desired input in a new stack, with a new memory and considering the environment of the execution of B. The execution of A waits for the execution of B to finish and to return a 0 or a 1 that signals unsuccessful or successful termination, and possibly some output, before resuming.

Conditional execution of code is possible due to the opcode `JUMPI`, whose operands are a boolean that indicates whether the jump should be performed or not and a position of the code to jump to in the affirmative case. There is also the unconditional `JUMP` to a specified position. There is an instruction, `JUMPDEST`, that only marks its position as a valid jump destination. If there is enough gas to execute it, a jump is valid if and only if it is to a valid jump destination. In particular, it is possible to jump back in a program and to create unstructured loops. In Chapter 4 we will develop the subject of estimating the gas cost of a program and perform a deeper analysis on code structure and loops.

Since the release of the first specification, a few more opcodes have been developed and integrated in the EVM. Grishchenko et al. ([6]) formalized the semantics of the original instructions; we extend their formalizations to the new operations `REVERT`, `RETURNDATASIZE`, and `RETURNDATACOPY`.

2.4 Execution model

The design of the virtual machine motivates the formal definition of the structures *machine state* and *execution environment*. The execution of a contract A with some input parameters may trigger the execution of a contract B – functions may call other functions. For analysis purposes, it is convenient to distinguish between elements that are relative to the execution of instructions of the code of the contract A and elements that are relative to the entire transaction that starts with the execution of A, including instructions of other contracts that were called by A.

The machine state and the execution environment only concern the execution of a contract. The machine state models the elements whose values are changed by executing instructions of the contract and the execution environment models the components that are constant during that execution.

Definition 6 (Machine state)**.** *The machine state is a tuple* $\mu = (gas, pc, m, i, s, rd)$ *such that:*

- $gas \in \mathbb{N}_{256}$ *is the available gas for the execution;*
- $pc \in \mathbb{N}_{256}$ *is the program counter;*
- $m \in \mathbb{B}^*$ *is the memory of the execution;*
- $i \in \mathbb{N}_{256}$ *is the number of active words in memory;*
- $s \in (\{0,1\}^{256})^{1024}$ *is the execution stack;*
- $rd \in \mathbb{B}^*$ *is the return data.*

In particular, when a contract A calls a contract B, B is run in a fresh execution stack and has a fresh memory. The initial available gas in the machine state of B is specified by A and cannot be more than the available gas in the machine state of A when the execution of B started. The program counter

starts at zero, so the code of contract B is executed from the beginning and continues to do so according to the input that was given to it.

Since the execution stack is the fundamental data structure of the EVM, we may refer to it simply as stack. This should not be confused with the call stack, a concept that will be introduced later. We will clearly distinguish between both stacks when that is not evident from the context.

The return data is not a part of the original definition of [6], but we include it here since we also consider the opcodes `RETURNDATACOPY` and `RETURNDATASIZE`. rd is a bytearray containing the output of the last call to other contract performed by the current transaction through one of the opcodes `CREATE`, `CALL`, `CALLCODE` or `DELEGATECALL`. If the output of the last call was empty, we write the content of rd as ε.

Definition 7 (Execution environment)**.** *The execution environment is a tuple* $\iota = (actor, input, sender, value, code)$ *such that:*

- $actor \in \mathbb{A}$ *is the address of the account that is currently executing;*
- $input \in \mathbb{B}^*$ *is the data given as input to the execution of the current contract;*
- $sender \in \mathbb{A}$ *is the address of the account that directly triggered the execution of the current contract;*
- $value \in \mathbb{N}_{256}$ *is the value transferred in the current execution;*
- $code \in \mathbb{B}^*$ *is the code currently executing.*

On the other hand, the transaction effect contains information about the entire transaction that is updated during the execution, but that only has effect when it finishes, and the transaction environment is the list of parameters that remain constant during the transaction execution. Notice that the transaction environment is set by the sender and by the executor, so it is not a constant property of the transaction itself.

Definition 8 (Transaction effect)**.** *The transaction effect consists of a tuple* $\eta = (bal_r, L, S_\dagger)$ *such that:*

- $bal_r \in \mathbb{N}_{256}$ *is the balance refund. It will be given to the beneficiary of the transaction once it is completed;*
- $L \in \mathbb{A} \times (\{\} \cup \mathbb{B}^{32} \cup (\mathbb{B}^{32})^2 \cup (\mathbb{B}^{32})^3 \cup (\mathbb{B}^{32})^4) \times \mathbb{B}^*$ *is the set of logs, used to store events during the transaction execution;*
- $S_\dagger \subseteq \mathbb{A}$ *is the suicide set, the set of addresses of accounts that received instructions to be deleted during the transaction execution.*

Definition 9 (Transaction environment)**.** *The transaction environment is a tuple* $\Gamma = (o, price, B)$ *such that:*

- $o \in \mathbb{A}$ *is the address of the account that started the transaction;*
- $price \in \mathbb{N}_{256}$ *is the gas price of the transaction;*

- $B \in \mathcal{B}$ *is the block that contains the transaction.*

A transaction is modelled using a *call stack*. An element of the call stack models the execution of a contract and it is of the form $(\mu, \iota, \sigma, \eta)$ – where μ is a machine state, ι is an execution environment, σ is a global state, and η is a transaction effect –, or EXC, or $HALT(\sigma, g, d, \eta)$ – where σ and η are as before, $g \in \mathbb{N}_{256}$ is the remaining gas and $d \in \mathbb{B}^*$ is the output data –, or $REV(g, d)$ – where g and d are as before. Elements of the types EXC, $HALT$, and REV can only occur on top of the call stack: their presence indicates termination of an execution. An element is added to the call stack when an inner execution begins, and an element is popped from the call stack when an inner execution finishes. The top of the call stack always represents the current execution. The call stack must not have more than 1024 elements – although Wood did not formally define the concept of call stack, he stated that if the call depth limit, which corresponds to the size of the call stack, exceeds 1024, then the execution should end with an exception. $HALT$ denotes regular halting, REV denotes an unsuccessful termination where the remaining gas is refunded to the caller account, and EXC denotes unsuccessful halting without refunding. The grammar for the call stacks is summarized in Figure 2.1. The domains of μ, ι, σ and η were already described in the respective definitions and are omitted here to ease the readability.

$$
\begin{aligned}
S &:= HALT(\sigma, g, d, \eta) :: S_{plain} \mid REV(g, d) :: S_{plain} \mid EXC :: S_{plain} \mid S_{plain} \\
S_{plain} &:= (\mu, \iota, \sigma, \eta) :: S_{plain} \\
\mu &:= (gas, pc, m, i, s, rd) \\
\iota &:= (actor, input, sender, value, code) \\
\eta &:= (bal_r, L, S_\dagger)
\end{aligned}
$$

Figure 2.1: Grammar for a call stack S

One of the main goals of this project is to develop a tool that is able to formally analyse smart contracts. More specifically, we would like to state a property about the variables involved in a transaction and check if that property is true after the execution of the transaction, regardless of the values given as input data.

Semantic methods intend to capture the meaning of a computer program; they are designed to formalize the general notion of "specification" that often lacks precision and clarity. There are several possible approaches to this idea; in the present work we will focus on structural operational semantics. This field of study breaks a computation into steps, the instructions of the program, and defines small-step rules that precisely express the changes in the state of the system performed by each instruction. For a more detailed study on semantic analysis, we refer the interested reader to [14] and [7].

As a consequence, a model of the system is required. The components of the system were formally defined in the previous section. For better readability, we will use the dot notation $\alpha.\mathtt{c}$ to refer to the component c of the tuple $\alpha \in \{\mu, \iota, \eta, \sigma(a)\}$, where $a \in \mathbb{A}$. The empty account $(0, \varepsilon, 0_F, 0)$ is denoted by the symbol $\perp$. The small-step rules that specify the changes in the components due to the execution of

each opcode described in [19] were formally defined in [6]. The opcodes `REVERT`, `RETURNDATACOPY` and `RETURNDATASIZE` were developed and became a part of the EVM later. We extended the semantics to contain the rules that model those opcodes as well; the semantics of the first opcode are presented in the following chapter and the semantics of the remaining two can be found in Appendix A.

A transaction is initiated by an external message: an action of an externally owned account. The transaction $T = (from, to, n, v, d, g, p)$ contains all the information needed to initialise the call stack, and the components o and *price* of the transaction environment; the last component B depends on the block where the transaction is being mined. The transaction environment is therefore $\Gamma = (from, p, B)$. The initial call stack contains a single element $(\mu, \iota, \sigma, \eta)$ such that:

- $\mu = (g, 0, \varepsilon, 0, \varepsilon, \varepsilon)$;
- $\iota = (to, d, from, v, \sigma(to).\texttt{code})$;
- σ is the global state function in the beginning of the execution;
- $\eta = (0, \emptyset, \emptyset)$.

From that point on, the call stack evolves according to the rules that model each opcode. In order to write the rules, it is necessary to introduce the notation for current opcode, $\omega_{\mu,\iota}$, depending on parameters *pc* and *code* from μ and ι, respectively:

$$\omega_{\mu,\iota} = \begin{cases} \iota.\texttt{code}[\mu.\texttt{pc}] & \text{if } \mu.\texttt{pc} \leq |\iota.\texttt{code}|; \\ \texttt{STOP} & \text{otherwise.} \end{cases} \tag{2.1}$$

As usual, $|\cdot|$ is the function that returns the size of a list.

We are now ready to introduce the small-step semantic rules that model the execution of a program on the Ethereum Virtual Machine. We will consider that every rule is composed by two parts: the premises, written at the top, and the consequences, written at the bottom, leading to the simple general structure sketched in Figure 2.2.

$$\frac{\omega_{\mu,\iota} = \texttt{OP} \qquad premises(S)}{\Gamma \models S \to S'}$$

Figure 2.2: General structure of a small-step rule

`OP` is an opcode. S, S' are call stacks. $premises(S)$ represents every condition on the elements of S that should hold for the rule to be applied. $\Gamma \models S \to S'$, where Γ is a transaction environment, should be interpreted as "a system under the transaction environment Γ, whose state is described by the call stack S, evolves to a state described by the call stack S'". As an example, one of the rules that model the `ADD` opcode is presented in Figure 2.3.

$$\frac{\omega_{\mu,\iota} = \texttt{ADD} \qquad valid(\mu.\texttt{gas}, 3, |s|) \qquad \mu.\texttt{s} = a :: b :: s \qquad \mu' = \mu[\texttt{gas}- = 3][\texttt{pc}+ = 1][\texttt{s} \to a + b \mod 2^{256} :: s]}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

Figure 2.3: Rule for the successful execution of the ADD opcode

$valid$: $\mathbb{N}_{256} \times \mathbb{N}_{256} \times \mathbb{N}_{10} \to \{0, 1\}$ is a function that checks if there is enough gas for the execution of the instruction and if the execution stack has the allowed size. It is defined by:

$$valid(x, y, z) = \begin{cases} 1 & \text{if } x \geq y \text{ and } z \leq 1024; \\ 0 & \text{otherwise.} \end{cases} \tag{2.2}$$

$valid(x, y, z)$ is a shortcut for $valid(x, y, z) = 1$, and $\neg valid(x, y, z)$ is a shortcut for $valid(x, y, z) = 0$. The notations $\alpha[\texttt{c}+ = x]$ and $\alpha[\texttt{c}- = x]$ are used to denote a tuple equal to α, where $\alpha \in \{\mu, \iota, \eta\}$, except that the component c is replaced by the current value of c incremented or decremented, respectively, by x units. Similarly, $\alpha[\texttt{c} \to x]$ represents a tuple equal to α but whose value of the component c was replaced by x. If the component c of α is a list, $\alpha.\texttt{c}[x, y]$ represent the contents of c from position x to position y, including both. Since the global state σ is not a tuple, but a function that can be seen as a set of key-value pairs, updating the account with address a in component c to the new value x is written $\sigma\langle a \to \sigma(a)[\texttt{c} \to x]\rangle$.

Therefore, the previous semantic rule means that if the current opcode is ADD, if the sender has enough gas and an allowed execution stack size, if the execution stack has at least two elements, and the first two elements of the execution stack are a and b, then, under the transaction environment Γ, the call stack transits from the state $(\mu, \iota, \sigma, \eta) :: S$ to the state $(\mu', \iota, \sigma, \eta) :: S$. μ' is equal to μ except that the available gas is decremented by 3 units, the program counter is incremented by 1 unit, and the stack has the first two elements popped and the element $a + b \mod 2^{256}$ pushed to it. The remaining components of the top element of the call stack are not changed, as well as the rest of the call stack. There are other rules for ADD that model the cases where the execution stack has less than two elements and where the available gas is less than 3 units, situations that result in unsuccessful termination.

When no rule can be applied, the call stack reaches a final configuration. Let the top element of the call stack be $(\mu, \iota, \sigma, \eta)$. σ becomes the global state function in that moment. The accounts whose addresses are in $\eta.\texttt{S}_\dagger$ are deleted, which means that for all $a \in \eta.\texttt{S}_\dagger$ the global state function is modified so that $\sigma(a) = \bot$. The miner that completed the proof of work for the block where the transaction is, *ben*, is rewarded through a process that involves the balance refunds $\eta.\texttt{bal}_\texttt{r}$ and a static block reward.

In the next chapter we will introduce the symbolic Ethereum Virtual Machine and detail some of the small-step rules that govern it.

Chapter 3

Formal verification of EVM bytecode

The main contribution of this work is the development of a symbolic Ethereum Virtual Machine. It can be used to simulate the execution of EVM bytecode containing symbolic features such as function parameters, transferred values, account balances, and memory and storage contents. The system replaces unknown values by symbols and stores and uses the assumptions made for each symbol. When the path that the execution should follow depends on a symbolic variable with an unknown value, every possibility is considered and the conditions on the value of the variable under which each path is taken are recorded. Each branch symbolically executes independently. In the end, every execution path is returned along with the final global state for each case. This symbolic machine was developed in Mathematica, taking advantage of its symbolic computation features.

Bytecode is very flexible as it can implement anything from simple arithmetic calculations to a structure similar to a class of object-oriented programming. Indeed, Solidity is an object-oriented language, influenced by C++, Python, and JavaScript, where the code of a smart contract is similar to a class and can have several attributes and methods. Usually, when a contract is called, what is meant is that a function of that contract is called with some input parameters, but, in general, the input given to a contract can be any sequence of bytes of any length. In order to develop a symbolic model of the EVM, some assumptions about the structure of the input had to be established and, in order to use it, some human action is needed to generate meaningful inputs. Therefore, the input specifies the function of the contract that should be called and the arguments. The symbolic and general nature of the model lies in the possibility of specifying symbolic arguments. In that case, the function is completely verified as it is run with symbols in place of the arguments. The output allows the user to check if there exists an execution path under which a property is violated.

We start by detailing the structure of every component of the symbolic EVM in Section 3.1, as well as the features and the limitations of our model. The semantics of this machine are explored in Section 3.2 through illustrative operations that show both the main functionalities of the EVM and the differences between the symbolic and the classic models. In Section 3.3 we study a real contract that was exploited and we are able to detect, from the results given by the symbolic machine, the execution path that contained the vulnerability.

3.1 Symbolic model

In spite of being based on the semantic rules of the EVM, a symbolic execution model of EVM bytecode requires a slightly different set of rules in order to encompass the possibility of branching. The state of the execution of a transaction cannot be modelled by a call stack anymore: it is now a list of call stacks, each representing a possible execution path.

Execution stack elements can be symbols, as well as account balances and transaction values, which allows the verification of properties about balances before and after transferences. These are among the most important kinds of properties: a user is typically interested in knowing if calling other contracts may cause him to lose money, or if transfering x ether to a contract may trigger the execution of code that takes more than x ether from his account. We analyse a contract assuming that we know the contracts that it may call or that may call it. Therefore, we state that a given property is valid or invalid in a fixed context. A valid property in an environment may be violated by an interaction with a contract from outside that environment. This should not be a problem since the code of every contract is publicly stored in the blockchain; however, one should keep in mind that new contracts are always being developed.

The model has some limitations: not every component of the machine state, global state, execution environment, transaction effect and transaction environment is allowed to be symbolic.

During an execution, $\mu.\texttt{gas}$ serves the main purpose of deciding if an operation may be executed or not. Later, it is used to determine the fee that should be paid to the miner. Since we are only interested in small-step execution, we may discard the computation of the fees. Given that, with $\mu.\texttt{gas}$ symbolic, in each operation there exists the possibility of an out-of gas exception and the possibility of normal execution, we decided to keep the gas numeric: there is no new information given by this branching. There is a downside to this option: a small class of programs for which not every case is taken into account. If the paths explicitly depend on the available gas, through the opcode `GAS`, when $\mu.\texttt{gas}$ is a number only one path will be taken. However, it is not common for a program to directly depend on the available gas and it is possible to test if that is the case in a simple way.

It does not make sense for components directly related to bytecode to be symbolic: $\mu.\texttt{pc}$, $\iota.\texttt{code}$, and $\sigma(a).\texttt{code}$ for all $a \in \mathbb{A}$.

Storage positions and contents may be symbolic, but it should be stressed that searching for a symbolic position $\mathbf{p}$ either succeeds, returning the contents of the position stored exactly as $\mathbf{p}$, or fails to find it, returning 0 by default as the content of position $\mathbf{p}$. In other words, if there is no information about the value stored in position $\mathbf{p}$ of the storage, we do not consider the possibility that $\mathbf{p}$ is an actual numerical position since, in that case, we would have to return the content of every position of the storage as a valid possibility.

The same is valid for account addresses: $a \in \mathbb{A}$, $\iota.\texttt{actor}$, $\iota.\texttt{sender}$, and $\Gamma.\texttt{o}$ may be symbols, but in order to access, for instance, the balance of a symbolic $\iota.\texttt{actor}$, the account with address $\iota.\texttt{actor}$ needs to be stored under that symbolic address, otherwise the symbolic execution assumes that the account does not exist (it is the empty account).

The model of the memory management makes some assumptions too. A number taken from the

execution stack is written with 256 bits and should be decomposed in 32 bytes to be saved in the memory. It is possible to access the memory in any position; in particular, it is possible to store a number x starting in position a of the memory, a number y starting in position $a + 32$ and to load later the number formed by the 32 bytes that start in position $a + k$ for any $k \in \mathbb{N}$ (`MLOAD` with argument $a + k$). If $0 < k < 32$, the number pushed to the stack as a result of the operation may be different from any number that was stored. It is a combination of the last digits of x and the first digits of y. Although this flexibility is not a problem by itself, it certainly does not promote transparency in a system that should be prepared to be formally verifiable.

If the element to store is a symbol, $\mathbf{x}$, the list of digits of $\mathbf{x}$ is not directly available. Most developers do not deal directly with the memory as they code in high-level languages. Compilers naturally store numbers in positions that are multiples of 32 (starting from 0) and, in the vast majority of the situations, the elements that are loaded correspond to elements that were previously stored. Given this context, we decided to represent the action of writing a symbolic element $\mathbf{x}$ in a numeric position p of the memory $\mu.\mathtt{m}$ with 32 copies of the symbol $\mathbf{x}$, each in a position from p to $p + 31$. Unlike storage positions, memory positions must be numeric. The difference is, again, due to the discrepancy between the structures. Since an element is stored simultaneously in several contiguous positions of the memory, and that memory positions to access are typically explicitly declared, the most natural option was to keep the positions numeric. We acknowledge that these are limitations of the model but we consider that the structure that was developed covers a great fraction of the smart contracts in the blockchain.

The representation of the input of a transaction, $\iota.\mathtt{input}$, is influenced by the structure of bytecode compiled from Solidity code. Input is highly dependent on the structure of the piece of code to be called. Solidity is an object-oriented programming language, which means that Solidity code consists essentially of functions that can receive zero or more arguments. When compiled, functions are represented by their signature. The signature of the function is a 4-byte number corresponding to the first 4 bytes of the hash of a value dependent on the name given to the function and on the types of its arguments. Code compiled from Solidity assumes that the first 4 bytes of the input are the hash of the function of the contract to be called and the following bytes are the encoding of the arguments. It would not make sense for the function selector to be symbolic, but the arguments of numeric type may be, and in that case they are represented in the same way as symbolic elements in the memory. If an argument of a function is an array, the number of elements of the array must be fixed but the arguments can be symbols. This option is justified considering that most developers write their smart contracts in Solidity. The EVM design introduces a special kind of function – the *fallback function* –, that is executed whenever the first 4 bytes of the input do not correspond to any of the functions of the contract. This includes the cases where the input is empty or less than 4 bytes long. Naturally, the symbolic machine also implements this behaviour, so it is possible to analyse the fallback function.

Finally, a new component should be added to model the conditions assumed about the symbols currently being used. For instance, if a `JUMPI` makes the program jump to a given numerical position depending on a variable $\mathbf{b}$ whose value is unknown, the symbolic execution model must be able to consider both execution options, storing the assumption $\mathbf{b} = 0$ in the one that does not jump and $\mathbf{b} \neq 0$ in the

one that jumps. If a transaction with symbolic value $\mathbf{v}$ is performed, the paths where the sender does not have enough balance, where the sender has enough balance and $\mathbf{v} \neq 0$, and where $\mathbf{v} = 0$ should be individually considered, since every case is possible and the outcomes are different: in the first case an exception is thrown, and in the other two cases the execution continues but the gas costs differ. Then, the definition of the machine state μ is updated to contain a new field **cond** holding a boolean formula over a set of symbols. EXC, $HALT$ and REV symbols are modified in order to return the condition associated to the call stack and are now $EXC(\mathbf{cond})$, $HALT(\sigma, g, d, \eta, \mathbf{cond})$ and $REV(g, d, \mathbf{cond})$. This formula is returned at the end of the execution and should characterize the execution path.

In order to check the validity of a property P after the execution of a contract A with input $\mathbf{d}$, the code of A with input $\mathbf{d}$ should be run in the symbolic analyser. The final state will be a list of n call stacks, each containing on top an element that specifies a condition φ_i, $1 \leq i \leq n$. These conditions should be compared against P in order to determine if P is always respected or not.

As a final remark to this overview, it should be mentioned that the growth of the list of call stacks during the symbolic execution of a program revealed itself to be a mere theoretical concern. Although it is possible for a program to fork in every instruction, doubling the number of call stacks, and leading to a final number of call stacks that is exponential in the size of the program, in real contracts there is very few branching and the final number of call stacks is reasonable and smaller than the number of instructions of the program.

A concrete property will be explored in Section 3.3 in the context of the analysis of a real smart contract. In order to perform that analysis, it is necessary to introduce the semantic rules of each instruction, which will be done in the next section.

3.2 Semantics of the symbolic EVM

Before describing the semantic rules of the symbolic Ethereum Virtual Machine it is necessary to clarify what is meant by *condition* or *formula* associated to an execution path. In the following subsection we formally define the signature where we will be working from now on.

3.2.1 Signature

Let $\mathcal{S} \supseteq \mathbb{N}_{256}$ be an infinite and enumerable set. $\mathcal{S}$ is the set of symbols. We will write in **bold** the elements that can be instantiated with a symbol that is not a natural number, and in regular math font numeric elements and symbolic elements whose value is uniquely determined by **cond**. When an element that can be instantiated with a symbol is a natural number and the current operation forks, all branches but one will have **false** conditions. That is not a problem since that, at the end of the execution, every call stack whose condition on the top element evaluates to **false** is discarded. That branch corresponds to an impossible execution and, so, it is not of interest to our analysis. We will also denote in **bold** a list (or an array) where at least one of the elements is a possible symbol.

Using the notation from [15], we also consider a set $\mathcal{F}$ of function symbols and a set $\mathcal{P}$ of predicate symbols:

$$\mathcal{F} = \{+, -, \times, \div, \lfloor\ \rfloor, exp, mod, TC, \&, ||, \oplus, sdiv, smod, byte\} \tag{3.1}$$

$$\mathcal{P} = \{<, \leq, >, \geq, =, \neq, isAddress, isBool\} \tag{3.2}$$

$+, -, \times, \div, <, \leq, >, \geq, =$, and $\neq$ have arity 2 and their semantics are as expected. $\lfloor\ \rfloor$ has arity 1 and it is the mathematical floor. $exp(a, b) = a^b$ is the usual exponentiation operation and we will use the latter notation for better readability. $mod(x, y) = x \mod y$. $TC(x)$ is the two's complement representation of x. & is bitwise and, $||$ is bitwise or, and $\oplus$ is bitwise xor, and they have the expected semantics. *sdiv* and *smod* have arity 2 and correspond to the operations $\div$ and *mod*, respectively, where the arguments are interpreted as two's complement signed 256-bit integers. The last function symbol, *byte*, has arity 2 and $byte(a, b)$ is the bth byte of the 256-bit number a considering big endian notation where the first byte is byte number 0. Function symbols applied to arguments stay unevaluated when at least one argument is symbolic and there is no possible simplification considering $\mu.\texttt{cond}$.

The predicate symbols $isAddress$ and $isBool$ are intended to simplify symbolic bitwise operations. In compiled code from Solidity, when a given argument a of a function of the contract is of the type `address`, it is common to do the operation bitwise and (&) with arguments a and `0xF...F` (the hexadecimal number with 20 bytes equal to `FF`) in order to discard any additional bytes that the 32-byte number a taken from the execution stack might have. If a is known to be an address before this operation, or even in the beginning of the execution, that is, if $isAddress(a)$ is one of the conditions, then $\&(a,$ `0xF...F`$)$ can be simplified to a, which may be useful in further calculations. The same reasoning applies to $isBool$ when a symbolic value is known to be a boolean variable.

$$isAddress(x) = \begin{cases} \textbf{true}, & \text{if } x \leq 2^{160} - 1; \\ \textbf{false}, & \text{otherwise.} \end{cases} \tag{3.3}$$

$$isBool(x) = \begin{cases} \textbf{true}, & \text{if } x \leq 1; \\ \textbf{false}, & \text{otherwise.} \end{cases} \tag{3.4}$$

Having the signature $\Sigma = (\mathcal{F}, \mathcal{P}, \tau)$ of the symbolic EVM defined, where $\tau \colon \mathcal{F} \cup \mathcal{P} \to \{1, 2\}$ is the arity function informally described above, we may inductively construct the set of terms over Σ and $\mathcal{S}$, $T_\Sigma(\mathcal{S})$:

- $\mathcal{S} \subseteq T_\Sigma(\mathcal{S})$;
- If $x \in T_\Sigma(\mathcal{S})$, and if $f \in \mathcal{F}$ has arity 1, then $f(x) \in T_\Sigma(\mathcal{S})$;
- If $x, y \in T_\Sigma(\mathcal{S})$, and if $f \in \mathcal{F}$ has arity 2, then $f(x, y) \in T_\Sigma(\mathcal{S})$.

We are now finally able to define a set of formulas over Σ and $\mathcal{S}$, $L_\Sigma(\mathcal{S})$.

- **true**, **false** $\in L_\Sigma(\mathcal{S})$;

- If $x \in T_\Sigma(\mathcal{S})$, and if $p \in \mathcal{P}$ has arity 1, then $p(x) \in L_\Sigma(\mathcal{S})$;
- If $x, y \in T_\Sigma(\mathcal{S})$, and if $p \in \mathcal{P}$ has arity 2, then $p(x, y) \in L_\Sigma(\mathcal{S})$;
- If $\varphi_1, \varphi_2 \in L_\Sigma(\mathcal{S})$, then $\varphi_1 \wedge \varphi_2 \in L_\Sigma(\mathcal{S})$.

$L_\Sigma(\mathcal{S})$ is the domain of the component **cond** of the machine state μ of a symbolic EVM. In this context it suffices to consider conjunction of formulas since **cond** is meant to contain every assumption made about symbolic variables during the execution, and an assumption is a predicate symbol applied to one or two terms. We do not need to use negation since we can write the negation of every comparison symbol explicitly, and the negation of *isAddress* and *isBool* is of no practical use.

3.2.2 Small-step rules

The general structure of a rule changes to encompass multiple call stacks. Given that the universe is now a list of call stacks, a step consists of the execution of an instruction in each call stack, namely, the instruction in the position that corresponds to the program counter of the machine in each of the top elements of the call stacks. We will write the list of call stacks vertically for better readability. If there are n call stacks, the evolution of the call stack $(\mu_i, \iota_i, \sigma_i, \eta_i) :: S_i$ is described by the rule that is applicable to the opcode $\texttt{OP}_i$ in the conditions $premises(S_i)$, where $1 \leq i \leq n$, whose general structure is presented in Figure 3.1.

$$\frac{\begin{array}{ccc} \omega_{\mu_1,\iota_1} = \texttt{OP}_1 & \dots & \omega_{\mu_n,\iota_n} = \texttt{OP}_n \\ premises(S_1) & \dots & premises(S_n) \end{array}}{\Gamma \models \begin{array}{l} (\mu_1, \iota_1, \sigma_1, \eta_1) :: S_1 \\ \cdots \\ (\mu_n, \iota_n, \sigma_n, \eta_n) :: S_n \end{array} \rightarrow \begin{array}{l} (\mu'_1, \iota'_1, \sigma'_1, \eta'_1) :: S'_1 \\ \cdots \\ (\mu'_{n'}, \iota'_{n'}, \sigma'_{n'}, \eta'_{n'}) :: S'_{n'} \end{array}}$$

Figure 3.1: General structure of a small-step rule in the symbolic machine

Since the executions of the n call stacks are independent from each other, in the following we will only write one call stack on the left hand side and the call stacks that result from that one on the right hand side. The number of call stacks may increase after a step: a call stack may fork whenever an operation over elements of the execution stack does not return a numeric value due to the symbolic nature of one or more elements. The new list of call stacks is the union of the ones resulting from the individual execution of each call stack.

Initialisation is very similar to the numeric case: the only modification is in the initial value of the machine state μ: it is now $(g, 0, \varepsilon, 0, \varepsilon, \varepsilon, \mathbf{cond})$, where $\mathbf{cond} \in L_\Sigma(\mathcal{S})$ is the initial condition that we wish to specify. Possible examples are statements about environmental variables that are known to be addresses, through *isAddress*, or about the transaction value. If there are no assumptions, **cond** is **true** by default.

During the symbolic execution we maintain a list of call stacks $S_1, \dots, S_n$. If there exists a call stack, S_i, $1 \leq i \leq n$, such that no rule can be further applied to it, we consider that its configuration in the

next step is equal to the current configuration. We say that we reach a final state when there is no rule that can be applied to any of the call stacks $S_1, \ldots, S_n$. Therefore, we are looking for a fixed point of the function that reads the current configuration and updates each call stack according to the current context. The finalisation should contain the payment of rewards to the miners and the destruction of the accounts that were signaled to be eliminated during the execution but, since we are interesting in modelling smart contract execution and not blockchain usage, we did not implement these functionalities. As it was already described, when the symbolic execution finishes our aim is to determine if a given property P holds.

In the following, we will present the most significant semantic rules of the symbolic Ethereum Virtual Machine. The rules should illustrate the main characteristic features of the EVM and the most important differences between the original and the symbolic machines.

Basic arithmetic instructions, comparison operators and stack operations. We start with the arithmetic operation `ADD` to ease the comparison with semantics of the regular EVM. The structure is the same, the only modification is in the nature of stack elements, that are now allowed to be symbols. The expression $\mathbf{a}+\mathbf{b} \mod 2^{256}$ should be resolved if $\mathbf{a}$ and $\mathbf{b}$ are numeric and be stored as $\mathbf{a}+\mathbf{b} \mod 2^{256}$ otherwise. The rule is presented in Figure 3.2.

$$\frac{\omega_{\mu,\iota} = \texttt{ADD} \qquad \mathit{valid}(\mu.\texttt{gas}, 3, |\mathbf{s}|) \qquad \mu.\mathbf{s} = \mathbf{a} :: \mathbf{b} :: \mathbf{s} \\ \mu' = \mu[\texttt{gas} - = 3][\texttt{pc} + = 1][\mathbf{s} \to \mathbf{a} + \mathbf{b} \mod 2^{256} :: \mathbf{s}]}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

Figure 3.2: Rule for the successful execution of the `ADD` opcode in the symbolic machine

The rules for comparison operators must be able to accomodate two possibilities. One call stack generates two call stacks, each equal to the previous one in every component but the machine state of the top element. Each machine state contains the conjunction of the previous condition with one of the new possible conditions – for example, the opcode `LT` with arguments $\mathbf{a}$ and $\mathbf{b}$ results in the new conditions $\mathbf{a} < \mathbf{b}$ and $\mathbf{a} \geq \mathbf{b}$.

Stack operations, like `PUSH`n, `POP`, `DUP`n and `SWAP`n, have the expected semantics: only the execution stack, the available gas and the program counter of the top element of the call stack are changed. Notice that it is not possible to `PUSH` a symbol to the stack since the operand of the `PUSH`n opcode is the number formed by the n bytes that follow the push instruction in the currently executing code. However, the execution stack may contain symbols that were added to it through requests for the value of the transaction, the content of a position of the storage, or the balance of a given account, among other examples, so `POP`, `DUP`n and `SWAP`n can handle symbols.

Terminating opcodes. `RETURN` is used to successfully terminate an execution with the possibility of returning output. The output is taken from the memory of the execution. The two operands of this instruction are the initial position of the memory and the size of the portion that should be returned. Both arguments must be numeric since they represent positions, but the memory contents that form the

output can contain symbols. The condition is returned in the $HALT$ element in order to be passed on to the outer execution or to be a part of the final state, as it can be seen in Figure 3.3.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{RETURN} \\ \mu.\mathbf{s} = io :: is :: \mathbf{s} \quad aw = M(\mu.\mathtt{i}, io, is) \quad c = C_{mem}(\mu.\mathtt{i}, aw) \\ valid(\mu.\mathtt{gas}, c, |\mathbf{s}|) \quad \mathbf{d} = \mu.\mathtt{m}[io, io + is - 1] \quad g = \mu.\mathtt{gas} - c\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to HALT(\sigma, g, \mathbf{d}, \eta, \mu.\mathtt{cond}) :: S}$$

Figure 3.3: Rule for the successful execution of the `RETURN` opcode in the symbolic machine

Memory usage consumes gas. $M\colon (\mathbb{N}_{256})^3 \to \mathbb{N}_{256}$ is the memory expansion function, used to determine the new number of active words in memory given the current number of active words in memory $\mu.\mathtt{i}$, and the offset io and the size is of the memory accessed by the current instruction. $C_{mem}\colon (\mathbb{N}_{256})^2 \to \mathbb{Z}$ is an auxiliary function used to compute gas costs due to the variation of memory usage. These functions are defined in Appendix A.

When the `RETURN` opcode is run, if the execution stack has at least two elements io and is, a size no greater than 1024, and if there is enough gas, then the call stack is popped and a new terminating element is pushed to it. $HALT(\sigma, g, \mathbf{d}, \eta, \mu.\mathtt{cond})$ contains the global state σ and the transaction effect η that were specified by the previous top element of the call stack, the available gas g – equal to the previous available gas minus the gas spent by this operation –, the return data $\mathbf{d}$, which is the content of the memory between the positions io and is, and the condition $\mu.\mathtt{cond}$ of the previous machine state. If the current execution was caused by a `CALL`-like opcode, in next step the changes in the components present in the $HALT$ element will become effective, including updating the return data to $\mathbf{d}$ and the available gas.

The `REVERT` opcode ends the current execution reverting all changes made during it. It can return some output from the memory, like `RETURN`, and the unused gas is refunded to the account that triggered the execution, unlike what happens when an exception is thrown. The main rule for `REVERT`, shown in Figure 3.4, is very similar to the main rule for `RETURN`, but σ and η are discarded since the modifications to the global state and to the transaction effect should not become effective.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{REVERT} \\ \mu.\mathbf{s} = io :: is :: \mathbf{s} \quad aw = M(\mu.\mathtt{i}, io, is) \quad c = C_{mem}(\mu.\mathtt{i}, aw) \\ valid(\mu.\mathtt{gas}, c, |\mathbf{s}|) \quad \mathbf{d} = \mu.\mathtt{m}[io, io + is - 1] \quad gas = \mu.\mathtt{gas} - c\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to REV(g, \mathbf{d}, \mu.\mathtt{cond}) :: S}$$

Figure 3.4: Rule for the successful execution of the `REVERT` opcode in the symbolic machine

Jumps. Recall that the set of valid jump destinations is the set of positions that contain the opcode `JUMPDEST`. Let $D\colon \mathbb{B}^* \to \mathbb{B}^*$ be the function that assigns each piece of code its set of valid jump destinations. The rules for the unconditional jump `JUMP` are similar to the slightly more complex rules for the conditional jump `JUMPI`. This opcode, when given a symbolic guard, causes the current call stack to split in two: one where the execution jumps to position i and other where it continues to the next

position. The conditions are updated in order to reflect the assumptions made in each case. The first argument cannot be symbolic since it represents a position of the code. The complete rule is exhibited in Figure 3.5.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{JUMPI} \quad valid(\mu.\texttt{gas}, 10, |\mathbf{s}|) \quad \mu.\mathbf{s} = i :: \mathbf{b} :: \mathbf{s} \quad i \in D(\iota.\texttt{code}) \\ \mu_1' = \mu[\texttt{pc}+ = 1][\mathbf{s} \to \mathbf{s}][\texttt{gas}- = 10][\texttt{cond} \to \mu.\texttt{cond} \wedge \mathbf{b} = 0] \\ \mu_2' = \mu[\texttt{pc} \to i][\mathbf{s} \to \mathbf{s}][\texttt{gas}- = 10][\texttt{cond} \to \mu.\texttt{cond} \wedge \mathbf{b} \neq 0]\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to \begin{array}{l}(\mu_1', \iota, \sigma, \eta) :: (\mu, \iota, \sigma, \eta) :: S \\ (\mu_2', \iota, \sigma, \eta) :: (\mu, \iota, \sigma, \eta) :: S\end{array}}$$

Figure 3.5: Rule for the successful execution of the `JUMPI` opcode in the symbolic machine

Storage management. Saving information in the storage of the currently executing account is done through `SSTORE`. This operation depends on two arguments that may be symbolic – position and content – and the gas cost varies according to the values of the arguments. Storing element b in position a, whose current content is x, has a cost of 20000 gas if $x = 0$ and $b \neq 0$ and of 5000 gas otherwise – it is more expensive to write a non-zero value in an empty position. Therefore, it is necessary to consider up to four cases due to the differences in the $\mu.\texttt{cond}$ field and in the remaining gas. In any successful case the function *stor* of the currently executing account should be modified so that $stor(a) = b$. We chose to present in Figure 3.6 the rule for the case where there is enough gas for the cheapest case but not for the most expensive.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{SSTORE} \quad \neg valid(\mu.\texttt{gas}, 20000, |\mathbf{s}|) \quad valid(\mu.\texttt{gas}, 5000, |\mathbf{s}|) \\ \mu.\mathbf{s} = \mathbf{a} :: \mathbf{b} :: \mathbf{s} \quad \mathbf{x} = (\sigma(\iota.\texttt{actor}).\texttt{stor})(\mathbf{a}) \quad \mathbf{cond} = \mu.\texttt{cond} \\ \mu_1' = \mu[\texttt{gas}- = 5000][\texttt{pc}+ = 1][\mathbf{s} \to \mathbf{s}][\texttt{cond} \to \mu.\texttt{cond} \wedge \mathbf{x} = 0 \wedge \mathbf{b} = 0] \\ \mu_3' = \mu[\texttt{gas}- = 5000][\texttt{pc}+ = 1][\mathbf{s} \to \mathbf{s}][\texttt{cond} \to \mu.\texttt{cond} \wedge \mathbf{x} \neq 0 \wedge \mathbf{b} = 0] \\ \mu_4' = \mu[\texttt{gas}- = 5000][\texttt{pc}+ = 1][\mathbf{s} \to \mathbf{s}][\texttt{cond} \to \mu.\texttt{cond} \wedge \mathbf{x} \neq 0 \wedge \mathbf{b} \neq 0] \\ \sigma' = \sigma\langle\iota.\texttt{actor} \to \iota.\texttt{actor}[\texttt{stor} \to \sigma(\iota.\texttt{actor}).\texttt{stor}[\mathbf{a} \to \mathbf{b}]]\rangle \quad \eta' = \eta[\texttt{bal}_\texttt{r}+ = 15000]\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to \begin{array}{l}(\mu_1', \iota, \sigma', \eta) :: S \\ EXC(\mathbf{cond} \wedge \mathbf{x} = 0 \wedge \mathbf{b} \neq 0) :: S \\ (\mu_3', \iota, \sigma', \eta') :: S \\ (\mu_4', \iota, \sigma', \eta) :: S\end{array}}$$

Figure 3.6: Rule for the execution of the `SSTORE` opcode in the symbolic machine where the available gas is between 5000 and 20000

Notice that the balance refund $\eta.\texttt{bal}_\texttt{r}$ only increases in the case where a previously written position is cleared. One of the call stacks records the exception associated with the corresponding condition. In the next step, each call stack will perform a step independently.

`SLOAD` accepts a symbolic argument, that corresponds to the position whose content should be retrieved, and pushes to the execution stack whatever is in that symbolic position. Naturally, since the symbolic machine is an isolated and experimental environment, a symbolic position of the storage only exists if it was stored during the same transaction.

Memory management. The semantics of the operation `MSTORE` reflect the assumptions made about the structure of the memory. It receives two arguments from the execution stack, a and b, and saves b in the memory as a 32-byte word: b is written in the 32 bytes from a to $a + 31$. a must be a number; b may be a symbol. If b is a symbol it is stored in a different way: 32 copies of the symbol are written in the desired portion of the memory.

$$\frac{\omega_{\mu,\iota} = \texttt{MSTORE} \qquad c = C_{mem}(\mu.\texttt{i}, aw) + 3 \\ valid(\mu.\texttt{gas}, c, |\mathbf{s}|) \qquad aw = M(\mu.\texttt{i}, a, 32) \qquad \mu.\mathbf{s} = a :: b :: \mathbf{s} \\ \mu' = \mu[\texttt{gas} - = c][\texttt{pc} + = 1][\texttt{m} \to \mu.\texttt{m}[[a, a + 31] \to b]][\texttt{i} \to aw][\texttt{s} \to \mathbf{s}]}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \texttt{MSTORE} \qquad c = C_{mem}(aw) - C_{mem}(\mu.\texttt{i}) + 3 \qquad valid(\mu.\texttt{gas}, c, |\mathbf{s}|) \\ aw = M(\mu.\texttt{i}, a, 32) \qquad \mu.\mathbf{s} = a :: \mathbf{b} :: \mathbf{s} \qquad \mathbf{b} \notin \mathbb{N}_{256} \\ \mu' = \mu[\texttt{gas} - = c][\texttt{pc} + = 1][\texttt{m} \to \mu.\texttt{m}[a \to \mathbf{b}] \ldots [a + 31 \to \mathbf{b}]][\texttt{i} \to aw][\texttt{s} \to \mathbf{s}]}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

Figure 3.7: Two rules that concern the execution of the `MSTORE` opcode in the symbolic machine

Environmental information. Many environmental opcodes are getter methods, so their semantics are trivial to derive. We detail the rules for the very important operation `CALLDATALOAD`. It receives a single argument, a position, and pushes to the execution stack the number formed by the 32 bytes starting in that position of the input $\iota.\texttt{input}$. Since it is assumed that the first four bytes of the input represent the function to be called, they must be numeric. The arguments of the function follow this reference, starting in positions that are congruent with 4 modulo 32. Each argument has a size of 32 bytes. If they are symbolic, they must be stored using 32 copies of the same symbol. The original specification states that this operation receives a numerical position a and returns the contents of $\iota.\texttt{input}$ from position a to position $a + 31$. The semantics of this operation in the symbolic machine result from a modification of the original ones in the following way:

- If $a = 0$, it checks if the first 4 bytes of $\iota.\texttt{input}$ are all numeric. If they are not, it raises an exception. If they are, it returns the number corresponding to those 4 bytes followed by 28 zeros. This option was motivated by the fact that the only common reason to execute `CALLDATALOAD` in position 0 is to get the hash of the function and, so, the last 28 bytes were going to be discarded in the next step anyway; there is no relevant information lost;
- If $a \neq 0$, we require that $pos \equiv 4 \mod 32$, otherwise an exception is thrown. If the bytes from a to $a + 31$ are all numeric, the instruction pushes the corresponding number to the stack. If every byte is the same symbol, it pushes that symbol to the stack. In any other case, an exception is thrown.

The rule in Figure 3.8 covers the case where $a = 0$ and d is a list of numbers. If the input is less than 4 bytes long, the element pushed to the stack is padded on the right with the necessary number of zeros so that it has 32 bytes.

$$\frac{\begin{array}{ccc} & \omega_{\mu,\iota} = \texttt{CALLDATALOAD} & \\ \mu.\mathbf{s} = 0 :: \mathbf{s} & valid(\mu.\texttt{gas}, 3, |\mathbf{s}| + 1) & k = min(|\iota.\texttt{input}|, 4) \\ d = \iota.\texttt{input}[0, k-1] & d' = d \cdot 0^{224+32-k} & \mu' = \mu[\texttt{gas}-=3][\texttt{pc}+=1][\mathbf{s} \to d' :: \mathbf{s}] \end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

Figure 3.8: Rule for the successful execution of the CALLDATALOAD opcode in the symbolic machine with argument equal to 0

If $\iota.\texttt{input}[0, k-1]$ contains a symbol, an exception is thrown. The rule in Figure 3.9 illustrates exceptional termination, with the symbol EXC being pushed to the call stack.

$$\frac{\begin{array}{ccc} & \omega_{\mu,\iota} = \texttt{CALLDATALOAD} & \\ \mu.\mathbf{s} = 0 :: \mathbf{s} & valid(\mu.\texttt{gas}, 3, |\mathbf{s}| + 1) & k = min(|\iota.\texttt{input}|, 4) \\ \mathbf{d} = \iota.\texttt{input}[0, k-1] & & \mathbf{d} \notin \mathbb{B} \end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to EXC(\mu.\texttt{cond}) :: S}$$

Figure 3.9: Rule for the unsuccessful execution of the CALLDATALOAD opcode in the symbolic machine with argument equal to 0

If the position is different from 0, the conditions that were previously stated are checked, as well as the size of the input. We only present here the rule where the request is successfully completed; cases where at least one byte is a symbol and the 32 bytes are not equal, where the input is too small, where $a \not\equiv 4 \mod 32$, where there is not enough gas or where the execution stack underflows raise exceptions and the rules are presented in Appendix A.

$$\frac{\begin{array}{cccc} & \omega_{\mu,\iota} = \texttt{CALLDATALOAD} & & \\ \mu.\mathbf{s} = a :: \mathbf{s} & a \equiv 4 \mod 32 & valid(\mu.\texttt{gas}, 3, |\mathbf{s}| + 1) & |\iota.\texttt{input}| \geq a + 31 \\ \iota.\texttt{input}[a] = \cdots = \iota.\texttt{input}[a+31] = \mathbf{d} & \mathbf{d} \notin \mathbb{B} & \mu' = \mu[\texttt{gas}-=3][\texttt{pc}+=1][\mathbf{s} \to \mathbf{d} :: \mathbf{s}] & \end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

Figure 3.10: One of the rules for the unsuccessful execution of the CALLDATALOAD opcode in the symbolic machine

Interactions between contracts. The previous examples illustrate the behaviour of the system during a simple execution. Naturally, only the top element of the call stack is changed. When a call to other account is performed, a new element is pushed to the call stack. CALL is one of the most complex opcodes: it starts an internal transaction. It takes seven values from the execution stack: gas to the inner execution, address to be called, value to be transferred, input offset, input size, output offset, and output size. The input of this transaction is taken from the memory of the caller and the output will be written in the memory of the caller, in the specified positions and with the specified sizes. Since a new element is pushed to the call stack, it is necessary to check that it will not overflow. The success of the operation also depends on the relationship between the balance of the sender and the transferred value, and on the

existence of enough gas. The gas cost depends on the value to be sent, on the existence of the called account, and on the gas specified by the sender. Since we do not allow the gas to be symbolic, we only need to care about the value to determine the possible execution paths. There are three possiblilities: the value is 0; the value is greater than the balance of the sender; or the value is different from zero and less than or equal to the balance of the sender. The second case triggers an exception by lack of funds; the first and the third cases need to be considered separately because the gas costs are different. In the third case we consider that the value is 1 in the auxiliary functions of the gas calculations because they return the same results for every non-zero value. The input may contain symbolic elements. The address of the recipient can be symbolic, but it should be noted that, if the stack element that represents the account is **to**, the account whose existence will be checked is $\mathbf{to} \mod 2^{160}$. It is possible to call this symbolic account if $\mathbf{to} \mod 2^{160}$ is an account of the global state σ or if the term $isAddress(\mathbf{to})$ is a part of the field $\mu.\mathtt{cond}$. The following rule, in Figure 3.11, models the case where the called account exists. If it did not exist, the gas cost would be higher.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathtt{CALL} \qquad \mu.\mathbf{s} = g :: \mathbf{to} :: \mathbf{va} :: io :: is :: oo :: os :: \mathbf{s} \qquad |S| + 1 \leq 1024 \\ \mathbf{to}_a = \mathbf{to} \mod 2^{160} \quad \sigma(\mathbf{to}_a) \neq \bot \quad aw = M(M(\mu.\mathtt{i}, io, is), oo, os) \quad \mathbf{d} = \mu.\mathtt{m}[io, io + is - 1] \\ c_{call_1} = C_{gascap}(0, 1, g, \mu.\mathtt{gas}) \qquad c_{call_2} = C_{gascap}(1, 1, g, \mu.\mathtt{gas}) \\ c_1 = C_{base}(0, 1) + C_{mem}(\mu.\mathtt{i}) + c_{call_1} \qquad c_2 = C_{base}(1, 1) + C_{mem}(\mu.\mathtt{i}) + c_{call_2} \\ valid(\mu.\mathtt{gas}, c_1, |\mathbf{s}| + 1) \qquad valid(\mu.\mathtt{gas}, c_2, |\mathbf{s}| + 1) \\ \sigma' = \sigma\langle \mathbf{to}_a \to \sigma(\mathbf{to}_a)[\mathtt{b} + = \mathbf{va}]\rangle\langle \iota.\mathtt{actor} \to \sigma(\iota.\mathtt{actor})[\mathtt{b} - = \mathbf{va}]\rangle \\ \mu'_1 = (c_{call_1}, 0, \lambda x.0, 0, \epsilon, \mu.\mathtt{rd}, \mu.\mathtt{cond} \wedge \mathbf{va} = 0) \\ \mu'_2 = (c_{call_2}, 0, \lambda x.0, 0, \epsilon, \mu.\mathtt{rd}, \mu.\mathtt{cond} \wedge 0 < \mathbf{va} \leq \sigma(\iota.\mathtt{actor}).\mathtt{b}) \\ \iota'_1 = \iota[\mathtt{actor} \to \mathbf{to}_a][\mathtt{input} \to \mathbf{d}][\mathtt{sender} \to \iota.\mathtt{actor}][\mathtt{value} \to 0][\mathtt{code} \to \sigma(\mathbf{to}_a).\mathtt{code}] \\ \iota'_2 = \iota[\mathtt{actor} \to \mathbf{to}_a][\mathtt{input} \to \mathbf{d}][\mathtt{sender} \to \iota.\mathtt{actor}][\mathtt{value} \to \mathbf{va}][\mathtt{code} \to \sigma(\mathbf{to}_a).\mathtt{code}]\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to \begin{array}{l}(\mu'_1, \iota'_1, \sigma, \eta) :: (\mu, \iota, \sigma, \eta) :: S \\ (\mu'_2, \iota'_2, \sigma', \eta) :: (\mu, \iota, \sigma, \eta) :: S \\ EXC(\mu.\mathtt{cond} \wedge \mathbf{va} > \sigma(\iota.\mathtt{actor}).\mathtt{b})) :: (\mu, \iota, \sigma, \eta) :: S\end{array}}$$

Figure 3.11: Rule for the successful execution of the `CALL` opcode in the symbolic machine where the called account exists

As a result, the current call stack is replaced by three, two of which correspond to regular configurations. The elements pushed to those call stacks are of the form $(\mu', \iota', \sigma', \eta)$. σ' is the same in both and it is equal to σ after the transference of the value **va** from the account $\iota.\mathtt{actor}$ to the account $\mathbf{to}_a$. The conversion $\mathbf{to}_a = \mathbf{to} \mod 2^{160}$ is needed because addresses are 160-bit numbers and execution stack elements are 256-bit numbers. ι'_1 and ι'_2 are the new execution environments: *actor* is the called contract $\mathbf{to}_a$ and *code* is its code, *sender* is the caller contract, *value* is either 0 or the symbol **va**, and *input* is the content of the memory of the caller between positions io and $io + is - 1$. The new machine states μ'_1, μ'_2 contain available gas c_{call_i} that depends on whether the value is null or not, program counter and number of active words in memory equal to 0, and empty memory and execution stack, differing only in the formulas $\mathbf{va} = 0$ and $\mathbf{va} \neq 0$ that they connect to their conditions.

$C_{base} \colon \mathbb{N}_{256} \times \{0, 1\} \to \mathbb{N}$ and $C_{gascap} \colon \mathbb{N}_{256} \times \{0, 1\} \times \mathbb{N}_{256} \times \mathbb{N}_{256} \to \mathbb{N}$ are functions that help to

compute the gas cost of a call operation and the gas that should be sent to the inner execution. These functions consider the available gas, the value, and a boolean value ($flag$) that indicates if the called account exists in order to determine the possible additional costs of the operation. C_{gascap} depends on the function $C_{extra}\colon \mathbb{N}_{256} \times \{0,1\} \to \mathbb{N}_{256}$. Their values are higher if there is ether to be transferred or if the called account does not exist. Having a fixed, known available gas $\mu.\mathtt{gas}$, c_1 and c_2 are fixed values, and if the second argument, $flag$, was 0 they would also be fixed, but different, values. C_{base}, C_{gascap}, and C_{extra} are formally defined in Appendix A.

c_i $(i = 1, 2)$ is the maximum amount of gas that can be spent in the inner execution, so it is checked that the caller has at least c_i available gas. We only subtract the gas used in the inner execution from the machine state of the caller when it finishes. This is not a problem since the execution of a transaction is linear: the caller does not execute or trigger the execution of any code while the called contract is still running.

In the next step, the execution of the transaction continues according to the top element of each call stack. The regular ones contains instructions to execute the code of $\mathbf{to}_a$ with input $\mathbf{d}$. The values of the components of μ', ι', σ', and η may change during this inner execution. If that ends successfully, the relevant changes (condition *cond* of the machine state μ, global state σ and transaction effect η) will be passed on to the second element of the call stack and the top element will be popped. Otherwise, the top element is simply popped and the modifications are not stored, including the transference of **va**. In any case the outer execution resumes from the position of the code where it stopped. The next rule models a situation where the inner execution of an existing contract started by the `CALL` opcode ended regularly, with $HALT$ on top of the call stack.

The new condition is written to the machine state. Output data may contain symbols. Gas costs can be computed from **va** because, if **va** is a symbol, it was already assumed in the execution of `CALL` a condition on **va** implying that it is equal to 0 or different from 0. This condition was stored in $\mu.\mathtt{cond}$ and can now be used by the symbolic analyser to simplify computations. In this case, depicted in Figure 3.12, the called account necessarily existed, otherwise the termination would have been exceptional.

$$\frac{\begin{array}{cc} \omega_{\mu,\iota} = \mathtt{CALL} & \mu.\mathtt{s} = g :: \mathbf{to} :: \mathbf{va} :: io :: is :: oo :: os :: \mathbf{s} \\ \mathbf{to}_a = \mathbf{to} \mod 2^{160} & aw = M(M(\mu.\mathtt{i}, io, is), oo, os) \\ c_{call} = C_{gascap}(\mathbf{va}, 1, g, \mu.\mathtt{gas}) & c = C_{base}(\mathbf{va}, 1) + C_{mem}(\mu.\mathtt{i}) + c_{call} \\ \multicolumn{2}{c}{\mu' = (\mu.\mathtt{gas} + g' - c, \mu.\mathtt{pc} + 1, \mu.\mathtt{m}[[oo, oo + os - 1] \to \mathbf{d}], aw, 1 :: \mu.\mathtt{s}, \mathbf{d}, \mathbf{cond})} \end{array}}{\Gamma \models HALT(\sigma', g', \mathbf{d}, \eta', \mathbf{cond}) :: (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma', \eta') :: S}$$

Figure 3.12: Rule for the successful return from a `CALL` opcode in the symbolic machine

The available gas in the new machine state μ' is finally reduced: it is equal to the previous one minus the gas sent, c, plus the gas returned, g'. The output $\mathbf{d}$ is written in the positions oo to $oo + os - 1$ of the memory of the caller. Symbolic outputs are written in the memory using os copies of the symbol. The values of the global state σ' and of the transaction effect η' at the end of the inner execution are written to the outer execution state. The execution environment ι of the caller does not change since it is relative to the call that started its execution. 1 is pushed to the execution stack of the caller to signal

successful termination.

Output size may not be previously known by the caller. If d is a numeric output with $k < os$ bytes, d is written in positions oo through $oo + k - 1$ of the memory and positions $oo + k$ through os keep their previous contents; if d is a numeric output with $k > os$ bytes, only the first os bytes get written in the memory. We interpret this design decision of Ethereum as a way of limiting the space of the memory of the caller that the inner execution can occupy. However, we question the clarity of the former situation, since it puts the caller in a situation where he may not know if the value of a byte in a position between oo and $oo+os-1$ was set by the output or by a previous instruction, and suggest the alternative of writing 0 in any proposed but unused slot of the memory. In 2017, Ethereum released the opcode `RETURNDATASIZE` that partially solved this problem, allowing the caller to know exactly the size of the output of the last `CALL`-like operation made in the current execution. The launch of this operation was accompanied by `RETURNDATACOPY`, that takes a position and a size from the execution stack and pushes the requested portion of the last output, $\mu.\mathtt{rd}$, to it. These opcodes have some similarities with `CALLDATACOPY` and `CALLDATASIZE` but operate over the output instead of the input. We formalised their semantics and they can be seen in Appendix A.

An execution can terminate unsuccessfully due to insufficient gas, insufficient number of elements in the execution stack to execute the current opcode, execution stack overflow, call stack overflow, lack of funds during a transaction, invalid jump destination, or invalid opcode. If an execution started by a `CALL` opcode ends with an exception, the behaviour of the symbolic analyser can be described by Figure 3.13. It is still assumed that the called account exists.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathtt{CALL} \\ \mu.\mathtt{s} = g :: \mathbf{to} :: \mathbf{va} :: io :: is :: oo :: os :: \mathbf{s} \qquad \mathbf{to}_a = \mathbf{to} \mod 2^{160} \\ \sigma(\mathbf{to}_a) \neq \bot \qquad aw = M(M(\mu.\mathtt{i}, io, is), oo, os) \\ c_{call} = C_{gascap}(\mathbf{va}, 1, g, \mu.\mathtt{gas}) \qquad c = C_{base}(\mathbf{va}, 1) + C_{mem}(\mu.\mathtt{i}) + c_{call} \\ \mu' = (\mu.\mathtt{gas} - c, \mu.\mathtt{pc} + 1, \mu.\mathtt{m}, aw, 0 :: \mathbf{s}, \varepsilon, \mathbf{cond})\end{array}}{\Gamma \models EXC(\mathbf{cond}) :: (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

Figure 3.13: Rule for the exceptional return from a `CALL` opcode in the symbolic machine

The only component of the element of the call stack relative to the execution to the caller that changes is the machine state. All gas sent to the execution of the called contract is lost. 0 is pushed to the execution stack to signal unsucessful termination.

If an execution triggered by a `CALL` ends with the `REVERT` opcode, a REV element is pushed to the call stack – Figure 3.14. The evolution of the system resembles some aspects of the termination with $HALT$ and some of the termination with EXC. The remaining gas is refunded to the caller and the output is returned, but 0 is written to the execution stack of the caller. The condition is updated.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{CALL} \\ \mu.\mathbf{s} = g :: \mathbf{to} :: \mathbf{va} :: io :: is :: oo :: os :: \mathbf{s} \qquad \mathbf{to}_a = \mathbf{to} \mod 2^{160} \\ \sigma(\mathbf{to}_a) \neq \bot \qquad aw = M(M(\mu.\mathtt{i}, io, is), oo, os) \\ c_{call} = C_{gascap}(\mathbf{va}, 1, g, \mu.\mathtt{gas}) \qquad c = C_{base}(\mathbf{va}, 1) + C_{mem}(\mu.\mathtt{i}) + c_{call} \\ \mu' = (\mu.\mathtt{gas} + g' - c, \mu.\mathtt{pc} + 1, \mu.\mathtt{m}[[oo, oo + os - 1] \to \mathbf{d}], aw, 0 :: \mathbf{s}, \mathbf{d}, \mathbf{cond})\end{array}}{\Gamma \models REV(g', \mathbf{d}, \mathbf{cond}) :: (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

Figure 3.14: Rule for the reverted return from a `CALL` opcode in the symbolic machine

It is also possible to call a contract using the opcodes `CALLCODE` or `DELEGATECALL`. The former executes the code of the called contract while keeping the field *actor* of the execution environment equal, and only uses the value as a guard for the execution, not actually transferring it. The latter is itself a variation of `CALLCODE` that has one less argument, value – no transference is made –, and that, besides the *actor*, also does not change the *sender* field of the execution environment ι.

Account creation. Creating an account with an address $a \in \mathbb{A}$ is any action that changes the value of $\sigma(a)$ from $(0, \varepsilon, 0_F, 0)$ to a tuple different from $(0, \varepsilon, 0_F, 0)$. In practice, there are three ways of creating an account. Outside the EVM, it is possible to interact with Ethereum through a client and create an externally owned account (without code). A private key and a public key are generated and the address of the new account is obtained from the `KECCAK-256` hash of the private key. It is also possible to create an account with a specific address $a \in \mathbb{A}$ that does not exist yet using a contract to transfer ether to the address a, either through a `CALL` or through a `SELFDESTRUCT` operation. This account, however, cannot be accessed by any user since it was not created by the adequate means. Finally, a contract may contain instructions to directly create an account, through the opcode `CREATE`. The function $newAddress\colon \mathbb{A} \times \mathbb{N}_{256} \to \mathbb{A}$ generates the address of the account using information about the creator: its address and its nonce.

$$newAddress(a, n) = \texttt{KECCAK-256}(RLP((a, n)))[96, 255] \tag{3.5}$$

$RLP\colon \mathbb{B}^* \to \mathbb{B}^*$ is an encoding function fully defined in [19]. The notation $[96, 255]$ means that only the last 160 bits of the resultant 256-bit hash are considered so that the new address has the desired size in bytes.

The `CREATE` operation takes three arguments: the value to be transferred to the new account, and the offset and the size of the portion of the memory that contains the initialisation code for the new account. The rule on Figure 3.15 is relative to the situation where the output of $newAddress(\iota.\mathtt{actor}, \sigma(\iota.\mathtt{actor.n}))$ is a number that is not the address of any existing account.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{CREATE} \\ \mu.\mathbf{s} = \mathbf{va} :: io :: is :: \mathbf{s} \quad aw = M(\mu.\texttt{i}, io, is) \quad c = C_{mem}(\mu.\texttt{i}, aw) + 32000 \\ valid(\mu.\texttt{gas}, c, |\mathbf{s}| + 1) \quad \rho = newAddress(\iota.\texttt{actor}, \sigma(\iota.\texttt{actor}).\texttt{n}) \quad \sigma(\rho) = \bot \\ |S| + 1 \leq 1024 \quad \mu' = (L(\mu.\texttt{gas} - c), 0, \lambda x.0, 0, \epsilon, \epsilon, \mu.\texttt{cond}) \\ i = \mu.\texttt{m}[io, io + is - 1] \quad \iota' = (\rho, i, \iota.\texttt{actor}, \mathbf{va}, \epsilon) \\ \sigma' = \sigma\langle\rho \to (0, \mathbf{va}, \epsilon, \lambda x.0)\rangle\langle\iota.\texttt{actor} \to \sigma(\iota.\texttt{actor})[\texttt{b}- = \mathbf{va}][\texttt{n}+ = 1]\rangle\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to \begin{array}{l}(\mu', \iota', \sigma', \eta) :: (\mu, \iota, \sigma, \eta) :: S \\ EXC(\mu.\texttt{cond} \wedge \mathbf{va} > \sigma(\iota.\texttt{actor}).\mathbf{b}) :: (\mu, \iota, \sigma, \eta) :: S\end{array}}$$

Figure 3.15: Rule for the successful execution of the CREATE opcode in the symbolic machine

We can see that this operation causes the symbolic machine to consider two possibilities. One of them models the case where the creator does not have enough balance to send **va**. The other one represents the beginning of a new execution, namely, the execution of the code $i = \mu.\texttt{m}[io, io + is - 1]$. The global state is updated with the subtraction of **va** from the balance of $\iota.\texttt{actor}$, the increment of the nonce of the creator, and the state of the account ρ. $L\colon \mathbb{N}_{256} \to \mathbb{N}_{256}$ is such that $L(n) = \left\lfloor \frac{n}{64} \right\rfloor$ and it is used to compute the gas sent to the initialisation of the new contract. If this execution finishes regularly, the field *code* of the account ρ will be updated with its output afterwards.

It is possible that the result of $newAddress(\iota.\texttt{actor}, \sigma(\iota.\texttt{actor}).\texttt{n})$ is an address a such that the account $\sigma(a)$ already exists. In that case, its code, storage and nonce are cleared, but the balance is kept and added to **va** to form the new balance of the account. The final code will also be the result of executing $\mu.\texttt{m}[io, io + is - 1]$. The semantic rule that describes this case differs from the rule above only in the aspects that were just mentioned, so we omit it here and refer the reader to Appendix A for the complete list of rules. It should be mentioned again that address collisions have a negligible probability of occurring and there are no reports of such problems in the Ethereum blockchain.

Account deletion. SELFDESTRUCT puts the address of the account $\iota.\texttt{actor}$ (usually the currently executing account, unless a CALLCODE or DELEGATECALL triggered the execution) in the list of accounts that should be deleted in the end of the transaction, $\eta.\texttt{S}_\dagger$. Deleting an account a means turning $\sigma(a)$ into $(0, \varepsilon, 0_F, 0)$. The information that the account is no longer active is added to the blockchain, but its previous contents are still stored in the antecedent blocks, so it is not completely deleted. The owner of the account stops to be able to access it and this action is irreversible. Before deletion, the balance of the account is sent to an address that is the only argument of this opcode. After the execution of this instruction, it is still possible to call this address (as it is possible to call any 160-bit value), but the ether transferred will not be accessible by anyone – unless, by chance, the same address is generated in a later account creation.

The increase in the refund balance $\eta.\texttt{bal}_\texttt{r}$ and the smaller cost of SELFDESTRUCT relatively to a CALL with transference of a non-zero value are incentives to destroy an account that its owner has no intention of using again. Since accounts are not deleted immediately, it is checked that this account is not already in the suicide set $\eta.\texttt{S}_\dagger$ before adding 24000 gas to the refund balance that will be given to the beneficiary of the transaction.

Symbolic addresses are handled similarly to what happens with the CALL operation. The rule presented

in Figure 3.16 models the case where the recipient exists.

$$\frac{\begin{array}{ccc}\omega_{\mu,\iota} = \texttt{SELFDESTRUCT} & \mu.\mathbf{s} = a :: \mathbf{s} & \mathbf{a}' = \mathbf{a} \mod 2^{160} \\ \sigma(\mathbf{a}) \neq \bot & valid(\mu.\texttt{gas}, 5000, |\mathbf{s}|) & g = \mu.\texttt{gas} - 5000 \end{array} \\ \sigma' = \sigma\langle \iota.\texttt{actor} \to \sigma(\iota.\texttt{actor})[\texttt{b} \to 0]\rangle\langle \mathbf{a}' \to \sigma(\mathbf{a}')[\texttt{b} \ += \sigma(\iota.\texttt{actor}).\texttt{b}]\rangle \\ r = (\iota.\texttt{actor} \in \eta.\texttt{S}_\dagger) \ ? \ 24000 \ : \ 0 \\ \eta' = \eta[\texttt{S}_\dagger \to \eta.\texttt{S}_\dagger \cup \{\iota.\texttt{actor}\}][\texttt{bal}_\texttt{r} += r]}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to HALT(\sigma', g, \varepsilon, \eta') :: S}$$

Figure 3.16: Rule for the successful execution of the `SELFDESTRUCT` opcode in the symbolic machine

This operation stops the current execution. The information about the transference of $\sigma(\iota.\texttt{actor}).\texttt{b}$ is immediately written in the global state, but the full modification on $\sigma(\iota.\texttt{actor})$ is delayed. The remaining gas g and the transaction effect η' are saved and there is no output.

3.3 Vulnerability detection in a real contract

We will illustrate the functionalities of the symbolic Ethereum Virtual Machine using a real contract that was discovered to contain a vulnerability in April 2018. This contract implements a coin, or token, named BecToken. Implementing a token means creating an environment where it is possible to store and transfer tokens. The contract follows the standard protocol developed by Ethereum ERC20, which specifies the structure and the arguments of basic functions that a contract that implements tokens should contain, such as `balanceOf(address tokenOwner)`, `transfer(address to, uint token)`, or `totalSupply()`, and so it was seen as a reliable contract.

The contract BecToken[1] is very simple and only contains a constructor function; its main functionalities are inherited from other contracts. One of them is PausableToken, which contains the vulnerable function `batchTransfer`. The simplified source code of this function, written in Solidity, can be seen in Figure 3.17. The token environment contains, as a global variable, a structure similar to a dictionary, `balances`, that maps each address to its amount of tokens.

The function `batchTransfer` receives an array of addresses, `_receivers`, and an integer, `_value`, checks if there are between 1 and 20 receivers and, if so, updates `balances` in order to increment the contents of each entry that is an address of `_receivers` by the amount `_value`. It should be mentioned that this transference is limited to the scope of this contract and it is not a transaction between accounts; it concerns BecToken coins instead of ether coins. The auxiliary functions `add` and `sub` are inherited from a standard library named `SafeMath` that contains basic arithmetic operations modulo 2^{256}.

Since this function implements a transference, we are interested in verifying if the total amount of tokens is the same before and after the function execution. That is, if an account $a \in \mathbb{A}$ calls the function `batchTransfer` with the intent of transferring a value $x \in \mathbb{N}_{256}$ to the accounts $a_1, \ldots, a_n \in \mathbb{A}$, where $1 \leq n \leq 20$, then the quantity $\texttt{balances}[a] + \texttt{balances}[a_1] + \cdots + \texttt{balances}[a_n]$ should be invariant.

[1]The complete code of the contract can be found at `https://etherscan.io/address/0xc5d105e63711398af9bbff092d4b6769c82f793d#code`.

```
function batchTransfer(address [] _receivers, uint256 _value) public returns (bool) {
    uint cnt = _receivers.length;
    uint256 amount = uint256(cnt) * _value;
    require(cnt > 0 && cnt <= 20);
    require(_value > 0 && balances[msg.sender] >= amount);

    balances[msg.sender] = balances[msg.sender].sub(amount);
    for(uint i = 0; i < cnt; i++) {
        balances[_receivers[i]] = balances[_receivers[i]].add(_value);
    }
    return true;
  }
```

Figure 3.17: `batchTransfer` function code

In order to specify a valid input, one needs to know the size of the array `_receivers`. We will assume that it contains 2 symbolic addresses $\mathbf{r_1}, \mathbf{r_2}$.

Recall that an execution is triggered by a transaction, which is modeled by $T = (from, to, n, v, d, g, p)$. We will make the following assumptions about the transaction that causes the execution of `batchTransfer`:

- $from$ is the symbolic address of the transaction sender **symbSender**;
- to is the contract to call, denoted by the symbolic address **bec**;
- n is the nonce and it is not needed for the execution;
- v is the value of the transaction, which we will denote by the symbol **symbValue**. Notice that the transaction value is not the same as the value specified as the input of the function `batchTransfer`: the transaction value is transferred to the current contract before the execution of any code; the argument of the function is treated according to the function body;
- d, the input data, has a rather complex structure. It contains the hash of the signature of the function to call and the types of its arguments `batchTransfer(address[], uint256)`, followed by the symbolic arguments $\mathbf{r_1}$, $\mathbf{r_2}$, and $\mathbf{v}$, encoded following the Ethereum standard;
- g, the available gas, must be numeric and is set to 100000 in order to ensure that the execution does not end unsuccessfully due to lack of gas;
- p, the gas price, is the symbol **symbGasPrice**.

The values of the global variables of a contract, such as `balances`, are stored in the contract's internal storage and, so, are part of the global state of the system. The initial global state σ contains the accounts $\mathbf{r_1}$, $\mathbf{r_2}$, **bec**, **symbSender**, and **symbOrigin**. The latter is a symbolic address that represents the original sender of the transaction, that in the general case may not be the same as **symbSender**. Every

field of the contents of the accounts $\mathbf{r_1}$, $\mathbf{r_2}$, **symbSender**, and **symbOrigin** is symbolic and will not be relevant for the current execution. In the case of **bec**, its balance and nonce are symbolic, its code is the bytecode of the contract PausableToken[2], and its storage contains the information about the content of `balances`: `balances[`**symbSender**`]` is $\mathbf{bal_S}$, `balances[`$\mathbf{r_1}$`]` is $\mathbf{bal_1}$, and `balances[`$\mathbf{r_2}$`]` is $\mathbf{bal_2}$.

The symbolic machine also considers conditions over the symbols. The initial condition is set as:

$$\begin{gathered} isAddress(\mathbf{r_1}) \wedge isAddress(\mathbf{r_2}) \wedge isAddress(\mathbf{bec}) \wedge isAddress(\mathbf{symbSender}) \\ \wedge \\ 0 \leq \mathbf{bal_1} < 2^{256} \wedge 0 \leq \mathbf{bal_2} < 2^{256} \wedge 0 \leq \mathbf{bal_S} < 2^{256} \wedge 0 \leq \mathbf{v} < 2^{256} \\ \wedge \\ \mathbf{bal_1} \in \mathbb{Z} \wedge \mathbf{bal_2} \in \mathbb{Z} \wedge \mathbf{bal_S} \in \mathbb{Z} \wedge \mathbf{v} \in \mathbb{Z} \end{gathered} \tag{3.6}$$

Finally, every element of the transaction environment $\Gamma = (origin, gasprice, block)$ is symbolic and will not be relevant in what follows. The call stack is initialised with a single element $(\mu, \iota, \sigma, \eta)$, built from the transaction, the global state and the condition as specified in the previous chapter.

The desired property can now be stated more precisely. The initial value of the sum of the balances is $\mathbf{bal_S} + \mathbf{bal_1} + \mathbf{bal_2}$. The final value of each balance is the content of the corresponding position of the storage of the contract **bec** in the end of the execution. We will denote final values by adding a prime. Thus, the property that we wish to verify is:

$$\mathbf{bal_S} + \mathbf{bal_1} + \mathbf{bal_2} \stackrel{?}{=} \mathbf{bal'_S} + \mathbf{bal'_1} + \mathbf{bal'_2} \tag{3.7}$$

The symbolic machine finishes the execution returning 42 final configurations, 27 of which correspond to successful termination and the remaining 15 to exceptional behaviour. Since exceptions revert all changes made to the global state, the property trivially holds in these cases. The great number of possible execution paths is partly due to the variable cost of the operation `SSTORE`, which is invoked each time an entry of `balances` is changed. Indeed, there are only 8 distinct configurations for the final global state σ.

A careful analysis of the global states along with the conditions that lead to them shows that the real number of possibilities is even smaller. In 6 of these configurations, the final sum was determined to be equal to the initial one – there are 6 different returned configurations because the machine was distinguishing cases where certain variables were equal to 0 or different from 0. For both of the remaining final states, the simplification of the final balances yielded:

- $\mathbf{bal'_S} = \mathbf{bal_S} - (2 \times \mathbf{v} \mod 2^{256})$;
- $\mathbf{bal'_1} = (\mathbf{bal_1} + \mathbf{v}) \mod 2^{256}$;
- $\mathbf{bal'_2} = (\mathbf{bal_2} + \mathbf{v}) \mod 2^{256}$.

[2]The bytecode was obtained from the source code using the online compiler Remix: `https://remix.ethereum.org/`.

In this context, $a \mod 2^{256}$ should be read as the unique number $b \in \mathbb{N}$ such that $b \equiv a \mod 2^{256}$ and $0 \leq b < 2^{256}$. Therefore, the sum of the final balances, $\mathbf{bal'_S} + \mathbf{bal'_1} + \mathbf{bal'_2}$, is equal to:

$$\mathbf{bal_S} - (2 \times \mathbf{v} \mod 2^{256}) + (\mathbf{bal_1} + \mathbf{v} \mod 2^{256}) + (\mathbf{bal_2} + \mathbf{v} \mod 2^{256}) \tag{3.8}$$

It is clear that $\mathbf{bal_S} + \mathbf{bal_1} + \mathbf{bal_2} \equiv \mathbf{bal'_S} + \mathbf{bal'_1} + \mathbf{bal'_2} \mod 2^{256}$, but the stronger result we are looking for does not always hold.

Indeed, it is possible to make this sum increase considering $\mathbf{v}$ such that $2 \times \mathbf{v} \mod 2^{256}$ is as small as possible. In other words, when $\mathbf{v} = 2^{255}$, the final sum becomes $\mathbf{bal_S} + (\mathbf{bal_1} + 2^{255} \mod 2^{256}) + (\mathbf{bal_2} + 2^{255} \mod 2^{256})$. Assuming that the balances $\mathbf{bal_1}$ and $\mathbf{bal_2}$ are smaller than 2^{255} (a reasonable assumption given that the total supply of these tokens is around 7×10^9), the final sum is $\mathbf{bal_S} + \mathbf{bal_1} + \mathbf{bal_2} + 2^{256}$, which means that 2^{256} tokens were generated out of thin air. Checking the final balances again, it is possible to see that the sender did not lose any money and the balance of each receiver increased by 2^{255} tokens. This was precisely the attack that the contract suffered. It was quickly detected because it was a transference with an unusually high value. Looking at the source code in Figure 3.17 cautiously it is possible to detect that the flaw is in the second line of code: the variable `amount` is equal to the product of `cnt` and `_value`, which overflows if this product is greater than 2^{256}.

Arithmetic overflow is one of the types of vulnerabilities already identified by the Ethereum community in an attempt to reduce exploits to contracts resulting from misunderstandings of the developer about the language, but the need for formal verification persists since this list is only typically updated when an attack is discovered.

Chapter 4

Estimation of gas costs

Program execution has a real cost for the entity that triggers it. The cost in ether of executing a contract depends on the gas price of the transaction and on the gas cost of the operations performed. Gas prices are volatile and determined by the market, but gas costs are fixed and depend on the computational work that the execution requires. A user of the Ethereum blockchain would like to know the exact amount of gas that will be spent if he calls a certain contract. Since the costs of some operations depend on input parameters given to the program and on the global state of the system, in general it is only possible to estimate this cost.

It was already mentioned that calling a contact typically means calling a function of it. The main aim of this chapter is to develop a systematic procedure to estimate the gas cost of a function of a contract as precisely as possible using the symbolic EVM that was introduced in the previous chapter. Since this cost may depend on the input and on environmental parameters, we would like to return a list of possible costs along with the respective conditions.

Section 4.1 introduces the concepts of control flow analysis needed for the study of this problem. In Section 4.2 a theoretical approach is taken and an algorithm developed by Tarjan to determine if a control flow graph is reducible is analysed. Theory and practice are combined in Section 4.3, where the development of the tool that estimates the gas cost of a program is explored and compared with the previously existing tool that is part of the Solidity compiler Remix. Finally, its main features are illustrated in Section 4.4 through real smart contract examples.

4.1 Basic concepts of control flow analysis

As we have discussed in the previous chapter, a program may imply several execution paths. Executing a function of that program with a given input and a given environment, therefore, corresponds to taking one of those execution paths. If we wish to study the behaviour of the function, without specifying inputs or environmental conditions, then our focus will be on a subset of the execution paths of the program. In order to formalize this idea, we will start by recalling the definition of program given in Section 2.3:

Definition 10 (Program)**.** *A program P of the Ethereum Virtual Machine is a finite sequence of bytes.*

We will now define some general concepts of control flow analysis, along with the adaptations needed for the study of our particular problem. In general, informally, a *basic block* of a program P is a maximal sequence of contiguous instructions of P that are always executed consecutively. In the context of the Ethereum Virtual Machine, assuming that the execution does not run out of gas, it is easy to see that a basic block begins in the first instruction, or in a jump destination instruction, or immediately after a conditional jump; and it ends in the last instruction, or in a jump instruction, or immediately before a jump destination instruction, or in a terminating instruction (which includes invalid opcodes).

Definition 11 (Basic block)**.** *Let $P = \{i_1, \ldots, i_n\}$ be a program. A basic block of P is a sequence of consecutive instructions of P, $\{i_k, i_{k+1}, \ldots, i_l\}$, such that the following two conditions hold:*

- *$k = 1$, or $i_k =$ `JUMPDEST`, or $i_{k-1} =$ `JUMPI`;*
- *l is the first index after k such that: $l = n$, or $i_l \in \{$`JUMP`, `JUMPI`, `STOP`, `RETURN`, `REVERT`$\}$, or $i_{l+1} =$ `JUMPDEST`, or i_l is an invalid instruction.*

Note that this definition may not match the informal definition previously given since the condition of a conditional jump may be always true or always false. In this case, there are two "blocks" of code that are always executed consecutively, which means that they should be merged in a single basic block, but they remain separated as there is a `JUMPI` between them. This should not be a worry since, usually, real programs do not deliberately contain these trivial conditions, also known as opaque predicates. Observe that the definition suggests a simple linear-time algorithm to find all basic blocks of a program, which will be in fact used in the construction of the gas estimator.

Basic blocks are the fundamental units of the control flow graph, a very useful abstraction to reason about the behaviour of a program. Intuitively, a control flow graph models the possible execution paths of a program: it is a directed graph whose vertices are the basic blocks and whose edges represent jumps between basic blocks. The structure of the EVM allows dynamic jumps: when the `JUMP` opcode is executed, the destination is the instruction corresponding to the number currently on top of the stack. The same holds for one of the possible destinations of a `JUMPI`; the other is the next instruction. In particular, it is possible to write a program such that the destination of a `JUMP` is an input parameter. In the present section we will only consider programs where all jumps are static: the destination of any `JUMP` is unique and determined by the program, and the destinations of a `JUMPI` are the next instruction and a unique instruction determined by the program. It should be stressed that this is the case for the vast majority of the jumps found in smart contracts. The *jump destination* of a `JUMP` or `JUMPI` opcode is the number on top of the stack when the opcode is executed. Again, we formally define the concept of control flow graph in the context of the EVM but one should keep the general definition in mind in the following subsections.

Definition 12 (Control flow graph)**.** *The control flow graph (CFG) of a program P is a directed graph (V, E) such that V is the set of basic blocks of P, and E is the set of edges $(u, v) \in V^2$, where $u = \{u_1, \ldots, u_k\}$, $v = \{v_1, \ldots, v_l\}$ are such that:*

- *$u_k =$ `JUMP`, $v_1 =$ `JUMPDEST`, and v_1 is the jump destination of u_k;*

- u_k = *JUMPI*, v_1 = *JUMPDEST*, *and* v_1 *is the jump destination of* u_k*;*
- u_k = *JUMPI, and* v_1 *is the instruction that follows* u_k *in* P*.*

The basic block that contains the first instruction of the program is called the *entry node* of the graph. It is common to assume that the CFG is a weakly connected graph, since unreachable code (from the entry point of the program) is trivial to optimize; it just needs to be removed. We can assume without loss of generality that the entry node does not have ingoing edges; if there is a jump to the first instruction of the program it is trivial to obtain an equivalent program such that that does not occur. It follows that every vertex v, except the entry node, has indegree $deg^-(v) \geq 1$. In the case of the EVM, provided that we only allow static jumps, we can further state that every vertex v has outdegree $deg^+(v) \leq 2$ since a basic block can only end in three possible ways: terminating/invalid opcode (outdegree 0), unconditional jump (outdegree at most 1) or conditional jump (outdegree at most 2).

Given a program P, we can build the control flow graph of P statically if the jump destinations of every jump are pushed to the stack right before the `JUMP` or `JUMPI` opcode. For EVM programs, although most jumps are in these conditions, most programs contain a jump that is not. In this case we need dynamic analysis to determine the jump destination, a topic that will be covered in Section 4.3.

Moreover, even assuming that the control flow graph is available, one of the difficulties that arises when computing the gas cost of a program is the existence of loops. If an execution path does not contain loops, then the cost of executing it is the sum of the costs of the opcodes that constitute it – even if these costs are not constant, one is able to obtain an expression that can be resolved to a number given a particular input of the program. If the path does contain a loop, then, for each loop, it is necessary to know how many times the loop is executed and under which conditions in order to calculate the real cost.

Our next goal is to be able to identify loops from the bytecode, which is not straightforward since "loop" is not a well-defined concept. We will present an algorithm to find natural loops (intuitively, the result of compiling `for` and `while` loops) in EVM bytecode using a variation of an algorithm developed by Robert E. Tarjan [17].

4.2 Loop analysis

In the present section we will assume that there exists an oracle to compute the control flow graph of a program. In Section 4.3 the rationale behind the computation of the control flow graph of a program as a part of the tool that was developed is explored.

In order to analyse the control flow graph, we will use standard notions of graph theory. If $(u, v) \in E$, then u is the *source* and v is the *target* of the edge (u, v). If $u, v \in V$, a *path* from u to v, denoted $u \to v$, is a sequence of vertices $u, w_1, \ldots, w_n, v$ such that there is an edge between every two consecutive vertices: $(u, w_1), (w_1, w_2), \ldots, (w_{n-1}, w_n), (w_n, v) \in E$. A sequence with a single vertex is a path from the vertex to itself. When there is no path from u to v in a given graph we will write $u \not\to v$. A *circuit* is a path $u \to v$ with at least two vertices where $u = v$, or a path $v \to v$ such that $(v, v) \in E$. An *acyclic graph* is a

graph without circuits. A *subgraph* $G' = (V', E')$ of $G = (V, E)$ is a graph such that $V' \subseteq V$ and $E' \subseteq E$. The subgraph of $G = (V, E)$ *induced by* V' is the graph $G' = (V', E')$ where $E' = \{(x, y) \in E \colon x, y \in V'\}$.

Definition 13 (Directed tree)**.** *A directed tree is a tuple* (V, E, r) *where* (V, E) *is a directed graph,* $r \in V$ *is a designated vertex called root, and for each* $v \in V$ *there is exactly one path from* r *to* v*.*

For simplicity, and whenever that is clear from the context, we will simply use the term "tree" when talking about directed trees. We will also drop the reference to the root. It follows easily from this definition that every vertex $v \in V \setminus \{r\}$ has indegree $deg^-(v) = 1$. It is also clear that a directed tree does not have circuits (that is, it is a tree in the classical sense), since in that case there would exist multiple paths from r to any vertex in the circuit. A *subtree* is, as usual, a subgraph which is a tree.

Definition 14 (Spanning tree)**.** *Given a (directed) graph* $G = (V, E)$*, a spanning tree of* G *is a (directed) tree* $T = (V', E')$ *such that* $V' = V$ *and* $E' \subseteq E$*, that is, a tree whose vertices are the same as those of* G *and whose edges are edges of* G*.*

Notice that, while every graph has a spanning tree, that might not be unique. There are several algorithms designed to build a spanning tree of a given graph. We will focus our attention in a spanning tree built from a depth-first traversal of a graph, which we will call depth-first spanning tree, or DFST for short. Let $ord \colon V \to \{1, \ldots, |V|\}$ be the function that assigns each vertex its number in the order they were first visited by a depth-first traversal, or *preorder*. Algorithm 1 computes a depth-first spanning tree of a graph G and the function ord at the same time.

Algorithm 1 Depth-first spanning tree with preorder numbering

Input: control flow graph $G = (V, E)$ with entry node e

Output: DFST T and array ord

```
1: for v ∈ V do
2:     ord[v] = 0
3:     visited[v] = 0
4:     add v to T
5: end for
6: count = 1
7: stack = emptyStack
8: push(stack, e)
9: visited[e] = 1
10: ord[e] = count
11: while stack ≠ ∅ do
12:     w = top(stack)
13:     if there exists u such that (w, u) ∈ E and visited[u] = 0 then
14:         count = count + 1
15:         add (w, u) to T
16:         push(stack, u)
17:         visited[u] = 1
18:         ord[u] = count
19:     else
20:         pop(stack)
21:     end if
22: end while
23: return T and ord
```

It is easy to see that this algorithm computes a spanning tree of G with n vertices and m edges in $O(n+m)$ time and using $O(n+m)$ space. The algorithm can return different spanning trees of the same graph, but they all share some properties.

Lemma 1. *Let T be a DFST of a graph G. Then the edges of G can be partitioned in three sets:*

- *Cycle edges: (u, v) such that $v \to u$ is a path of T;*
- *Forward edges: (u, v) such that $u \to v$ is a path of T;*
- *Cross edges: (u, v) such that $u \not\to v$ in T, $v \not\to u$ in T and $ord(v) < ord(u)$.*

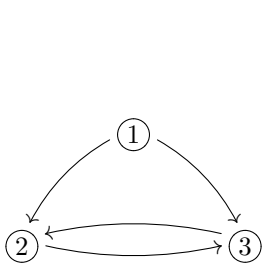
Proof. These sets are disjoint: it is clear that the third set does not intersect any of the others; cycle edges and forward edges are distinct since if (u, v) was both a cycle and a forward edge, then T would have a circuit.

These sets span every edge of G: suppose that (u, v) is not a cycle nor a forward edge. Then $u \not\to v$ and $v \not\to u$ in T. It remains to show that $ord(v) < ord(u)$. (u, v) is an edge of G which was not added to T. This means that, when the edges leaving u were processed, v was already a vertex of T, and so $ord(v) < ord(u)$. □

Let G be a directed graph and T a DFST of G. If there is a path from u to v in T, then u is an *ancestor* of v and v is a *descendant* of u. The terms *ancestor* and *descendant* are relative to a given spanning tree T but may be used in the context of a graph G when the tree that is being considered is fixed.

Definition 15 (Dominance)**.** *Let (V, E) be a CFG and let $u, v \in V$. u dominates v if every path from the entry node to v contains u.*

It is trivial from this definition that, if u dominates v, then u is an ancestor of v; the converse, however, is not always true, as it can be seen in Figure 4.1. Figure 4.1b shows that 2 is an ancestor of 3 in T and it is possible to reach 3 in G without passing through 2.



(a) Control flow graph G

(b) DFST T of G

Figure 4.1: A control flow graph and a possible DFST

An edge (u, v) such that v dominates u is called a *back edge*, and an edge (u, v) such that v is an ancestor of u was previously defined as a *cycle edge*. The last assertion can be reformulated in these terms by stating that a back edge is always a cycle edge, but the converse may be false. Our primary interest will be in graphs where every cycle edge is a back edge. One can test whether a CFG has this property by running Tarjan's algorithm, which checks if a graph is *reducible*. The equivalence between these two concepts will be proved right after the description of the procedure. Tarjan's algorithm consists of applying a "contraction" based on cycle edges to the CFG G until it is determined that G is irreducible or we run out of cycle edges – in which case we conclude that G is reducible. In the following, G is a CFG, V is the set of vertices of G, E is the set of edges of G and T is a depth-first spanning tree of G.

Definition 16 (Collapsing operation)**.** *Let $v, w \in V$. The result of collapsing w into v is the graph $G' = (V', E')$ such that:*

- $V' = V \setminus \{w\}$
- $E' = (E \setminus \{(z, w) \colon z \in V\}) \cup \{(v, z) \in V \times V \colon (w, z) \in E\} \cup \{(z, v) \in V \times V \colon (z, w) \in E\}$

The collapsing operation is particularly important when the collapsed vertex w has only one incident edge. This transformation is known in the literature (for instance [17]) as T_2.

Definition 17 (Transformation T_2)**.** *Transformation T_2 is an operation over a graph $G = (V, E)$ such that, if v, w are vertices such that $(v, w) \in E$ is the only edge incident in w, w is collapsed into v.*

Definition 18 (Reducible graph)**.** *A graph is reducible if the successive application of the transformation T_2 results in a graph with a single vertex.*

A naïve algorithm to successively apply T_2 until no transformation is possible would take $O(n^2)$ time in a graph with n vertices.

For every vertex $w \in V$ we define:

- $C(w) = \{v \in V \colon (v, w) \text{ is a cycle edge in } G\}$;
- $P(w) = \{v \in V \colon \exists z \in C(w) \text{ such that there is } v \to z \text{ in } G \text{ not containing } w\}$.

We observe that $C(w) \subseteq P(w)$: if $v \in C(w)$, then the trivial path v ensures that $v \in P(w)$. $C(w)$ is the set of sources of cycle edges ending in w; intuitively, it is the set of blocks where the program "jumps back". Therefore, if $C(w)$ is not empty, then w can be viewed as a possible loop header. Hence, $P(w)$ is the body of the loop if there are no jumps whose target is an instruction in the middle of the loop: it is the set of blocks that can reach the point where the program "jumps back" without passing through the loop header.

For efficiency reasons, Tarjan's algorithm represents the contractions of G through a *disjoint-set data structure*, also known as *union-find*. This data structure is used to model partitions of a set: in this case, the set of vertices of a graph G, V. Each subset of the partition has a representative element. Therefore we write a disjoint-set data structure of V as $D_V = \{(S_1, e_1), \ldots, (S_k, e_k)\}$, where $e_i \in S_i$ for every $1 \leq\ i \leq k$ and $\{S_1, \ldots, S_k\}$ is a partition of V. Let $\mathcal{D}_V$ be the set of all disjoint-set data structures of a set V, and $\mathcal{D}_V^*$ be the set of all disjoint-set data structures of all subsets of a set V. This data structure supports three operations.

makeset is used during the construction of the partition. If $x \in V$, it adds the subset $\{x\}$ of V to the current partition of a subset of V. The representative element is x.

$$\begin{aligned} makeset \colon V \times \mathcal{D}_V^* &\to \mathcal{D}_V^* \\ (x, D_V^*) &\mapsto D_V^* \cup (\{x\}, x) \end{aligned} \tag{4.1}$$

$find$ returns the representative element of the subset of the partition where x currently is. $find(x, D_V) = (y, D_V)$, where y is such that $x \in S$ and $(S, y) \in D_V$. This operation does not change the partition of V nor the representants. Some implementations, however – including the one that we will consider – change the internal representation of the partition, so we chose to keep a reference to the structure in the abstract definition.

$$\begin{aligned} find \colon V \times \mathcal{D}_V &\to V \times \mathcal{D}_V \\ (x, D_V) &\mapsto (y, D_V) \end{aligned} \tag{4.2}$$

union changes the data structure by replacing two subsets of the partition $S_1, S_2 \subseteq V$ by their union $S_1 \cup S_2 \subseteq V$.

$$\begin{aligned} union\colon V \times V \times \mathcal{D}_V &\to \mathcal{D}_V \\ (x, y, D_V) &\mapsto D'_V \end{aligned} \tag{4.3}$$

If $x, y \in V$ are such that $x \in S_1, y \in S_2$ and $(S_1, e_1), (S_2, e_2) \in D_V$, then $union(x, y, D_V)$ will return the updated data structure $D'_V = \{(S_1 \cup S_2, e_2)\} \cup D_V \setminus \{(S_1, e_1), (S_2, e_2)\}$. The representative element of the new subset can be chosen in many ways. For this algorithm, it is convenient to define it as the representative element of the second subset. The proof of the complexity of the algorithm assumes that each subset is implemented as a tree rooted at its representative element, and that the operation *union* merges two trees – using a heuristic known as *union by rank* – in a single tree whose root is chosen between the roots of the original trees according to their *rank*, a measure similar to tree height. This means that it is not guaranteed to be the root of the second tree. However, it is not hard to modify the data structure so that each root contains a reference to the node that should be the representative of its tree, and to update this reference to the representative of the second tree when a *union* is performed. This simple modification only adds a constant-time step to each operation, so the total time complexity bound is not affected.

4.2.1 Tarjan's reducibility test algorithm

Tarjan's algorithm is described in pseudocode in Algorithm 2. $ND(v)$ is the number of descendants of v in T. In the description of the algorithm and subsequent analysis we will drop the references to the data structure in the functions *makeset*, $find$, and *union* since that will always be clear. Initially, the data structure is empty; during the first for loop of Algorithm 2 the structure $\bigcup_{v \in V} (\{v\}, v)$ is built; after that it is only modified by the functions $find$ and *union* and it is assumed that they operate over the current structure in each call.

In the original description of the algorithm, Tarjan also considers *tree edges* in line 14. Tree edges are the edges of G that are also edges of T. Since these can also be classified as forward edges, we do not distinguish between them.

Proof of correction

We will now prove that this algorithm is correct. We follow the structure of the paper where the algorithm was presented [17] and write the proofs in more detail. In order to do that, we will need some auxiliary results. The following lemma allows us to check if there is a path between two vertices in T using the function ND:

Lemma 2. *For every $v, w \in G$, there exists a path $v \to w$ in T if and only if $ord(v) \leq ord(w) \leq ord(v) + ND(v)$.*

Proof. We start by noticing that the result holds trivially if $|V| = 1$. Assume now that $|V| > 1$. Suppose that $ord(v) \leq ord(w) \leq ord(v) + ND(v)$. Then w is first visited after the first visit to v. Between the

Algorithm 2 Tarjan's algorithm

Input: control flow graph $G = (V, E)$ such that $|V| = n$

construct a DFST of G, obtaining the preorder number $ord(v)$ of each vertex v, from 1 to n, and calculating $ND(v)$ for each vertex v
for v such that $ord(v) = 1$ until $ord(v) = n$ **do**
    make lists of cycle edges, forward edges and cross edges that enter v
    $makeset(v)$
end for
for w such that $ord(w) = n$ **step** -1 until $ord(w) = 1$ **do**
    $P = \emptyset$
    for each cycle edge (u, w) **do**
        add $find(u)$ to P
    end for
    $Q = P$
    while $Q \neq \emptyset$ **do**
        select a vertex x from Q and delete it from Q
        for each forward edge or cross edge (y, x) **do**
            $y' = find(y)$
            if $ord(w) > ord(y')$ or $ord(w) + ND(w) \leq ord(y')$ **then**
                return false
            end if
            if $y' \notin P$ and $y' \neq w$ **then**
                add y' to P and to Q
            end if
        end for
    end while
    for $x \in P$ **do**
        $union(x, w)$
    end for
end for
return true

first and the last visits to v, only descendants of v were visited, and every descendant of v was visited. This means that the value of *count* when v is pushed to the stack for the last time by Algorithm 1 is $ord(v) + ND(v)$. Then, w is in the subtree rooted at v, which means that there is a path from v to w in T.

Suppose now that there exists $v \to w$ in T. Then w is a descendant of v and, so, w belongs to the subtree rooted at v, which means that w will be visited between the first and the last times that v is pushed to the stack. The value of *count* when v is pushed to the stack for the last time is $ord(v)+ND(v)$. Therefore, $ord(v) \leq ord(w) \leq ord(v) + ND(v)$. □

The following results, that lead to Theorem 1, are attributed to Hecht and Ullman and until that point we follow the proofs in [9], after which we return to Tarjan's proof. We should note that the definition of reducibility used by Hecht and Ullman also considers the transformation T_1: If (v, v) is an edge of G, remove it from G. In this case, a graph is reducible if the successive application of T_1 and T_2, in any order, results in a graph with a single vertex. We will consider that, if G has edges of the form (v, v), they are removed before any other operation. Edges of the form (v, v) that are added to the graph by application of T_2 are ignored by Tarjan's algorithm when the vertices of P are merged into w. Therefore, the definitions can be seen as equivalent and the results obtained in both papers are compatible.

An important characterization of reducibility is given by a particular type of irreducible graph. The family of graphs represented in Figure 4.2 is known in the literature as (*). e is the entry node and can be the same node as a. The wavy arrows represent disjoint paths – paths that do not share nodes except possibly the first or the last. Formally, two paths $u_1, \ldots, u_k$, $v_1, \ldots, v_l$ are *disjoint* if $u_i \neq v_j$ for every $(i, j) \in \{1, \ldots, k\} \times \{1, \ldots, l\} \setminus \{(1, 1), (1, l), (k, 1), (k, l)\}$.

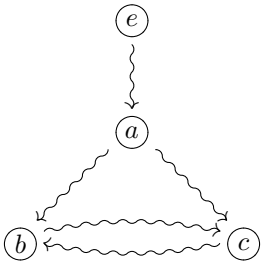


Figure 4.2: Irreducible graph (*)

Proposition 1. *G is irreducible if and only if it contains a graph of the family (*) as a subgraph.*

Proof. ($\Leftarrow$) Assume that G has a subgraph of the family (*). The proof is by induction on the number of vertices, n, which is at least 3.

Base: $n = 3$. Then G is the graph shown in Figure 4.1a (where $e = a$), or the same graph with the edge $(1, 2)$, or $(2, 3)$, or both. No transformation can be applied to any of those graphs, so they are all irreducible.

Induction hypothesis: If G is a graph with $n - 1$ vertices, where $n \geq 4$, with (*) as a subgraph, then G is irreducible.

Step: Let G be a graph with $n \geq 4$ vertices containing (*) as a subgraph. Suppose that G is reducible. Then there is a sequence of T_1, T_2 transformations that result in a graph with 1 vertex. At some point, the transformed graph G' has $n-1$ vertices. G' still contains (*) as a subgraph: the final edges of the paths $a \to b$, $a \to c$, $b \to c$, $c \to b$ cannot be contracted by T_2. Then, by induction hypothesis, G' is irreducible, which is a contradiction.

($\Rightarrow$) Suppose now that G is irreducible. The proof is, again, by induction on the number of vertices n.

Base: $n = 3$. Let e be the entry node and let a, b be the other two vertices of G. e does not have entering edges and there exist paths $e \to a$, $e \to b$. It is easy to verify that, if every vertex has at most one entering edge, the graph is reducible. If a and b have two entering edges each, then it is not possible to apply T_2 and G is irreducible. (*) is obviously a subgraph of this graph. If one vertex, a, has two entering edges and the other, b, has only one, we can apply T_2 and b is collapsed into a or into e. The resulting graph is always reducible.

Induction hypothesis: If $n \geq 4$ and the irreducible graph G has $k < n$ vertices, then (*) is a subgraph of G.

Step: Let G be an irreducible graph with $n \geq 4$ vertices. If T_2 can be applied to a vertex of G, the resulting graph G' is an irreducible graph with $n-1$ vertices, so by induction hypothesis (*) is a subgraph of G' and, so, of G. Assume now that it is not possible to apply T_2 to any vertex of G. Let T be a DFST of G and y the rightmost child of the entry node e in T. Since T_2 cannot be applied, y has two edges entering it in G: one from e and other from a vertex $x \neq e$. (x, y) is not a forward edge: $x \not\to y$ in T since y is a child of the root. By contradiction, if (x, y) is a cross edge, $ord(y) < ord(x)$. Consequently, since y is the rightmost child of the root, x is a descendant of y, and thus there is $y \to x$ in T. Yet this cannot hold by definition of cross edge. Then (x, y) is a cycle edge, and so there is $y \to x$ in T.

Consider the subgraph H of G induced by the vertices x, y, and all vertices z such that there exists $z \to x$ in G that does not contain y. If $e \in H$, then y does not dominate x. There exist an edge (e, y), a path $e \to x$ without y, an edge (x, y) and a path $y \to x$. Noticing that the paths are disjoint and that the edges of T are contained in the edges of G, we conclude that G has a subgraph of the family (*).

If $e \notin H$, then y dominates x. Furthermore, it dominates every vertex of H. Let $z \in H \setminus \{x\}$. If y did not dominate z, there would exist $e \to z$ not containing y and, so, $e \to x$ not containing y, which is a contradiction. Then H is a directed graph with entry node y and $k < n$ vertices and so by induction hypothesis it contains (*), so G also contains (*). □

This property, although useful to prove other results about reducible flow graphs, does not provide a direct algorithm to find out if a graph is reducible, since determining whether a graph has a given subgraph is an NP-complete problem.

Theorem 1. *G is reducible if and only if for every $v, w \in V$ such that (v, w) is a cycle edge, w dominates v in G.*

Proof. ($\Leftarrow$) Assume that G is reducible and that (v, w) is a cycle edge of G. If $v = w$ or if $w = e$, then w dominates v. Let us now assume that $v \neq w$ and $w \neq e$. Suppose that w does not dominate v. Then

there is a path $e \to v$ that does not contain w. Let p be this path, and let q be a path $e \to w$ and r be the path $w \to v$ that exists because (v, w) is a cycle edge. We can assume that q and r do not share edges.

Let $p \cap q$ be the set of vertices that are both in p and in q. This set is not empty: $e \in p \cap q$. Let a be the vertex of $p \cap q$ which is "closest" to w: there exists $a \to w$ containing k vertices and, if $a' \in p \cap q$, then all paths $a' \to w$ have k or more vertices. Similarly, let $p \cap r$ be the set of vertices that are both in p and in r and let b be the vertex of $p \cap r$ that is "closest" to w: there exists $w \to b$ containing k' vertices and, if $b' \in p \cap r$, then all paths $w \to b'$ have k' or more vertices. Then a, b, and w correspond to a, b, and c of the irreducible graph (*) shown in Figure 4.2. $a \to b$ and $a \to w$ are disjoint by definition of a. $a \to b$ and $w \to b$ are disjoint by definition of b. Then all paths are disjoint and (*) is a subgraph of G, so G is irreducible; contradiction.

($\Rightarrow$) Assume that G is irreducible. Then by Proposition 1 G has a graph of the family (*) as a subgraph. Let e, a, b and c be as in Figure 4.2, and let d be the vertex such that (d, b) is the last edge of the path $c \to b$. There exists a DFST T of G such that the paths $a \to b$, $b \to c$, and $c \to d$ are in T. (d, b) is a cycle edge since there is $b \to d$ in T. b does not dominate d: the concatenation of the paths $e \to a$, $a \to c$, and $c \to d$ yields a path from the entry point to d that does not contain b. □

Lemma 3. *G is reducible if and only if for all w and for all $v \in P(w)$, there exists $w \to v$ in T.*

Proof. ($\Leftarrow$) Assume that G is not reducible. Then by Theorem 1 there exists a cycle edge (v, w) in G such that w does not dominate v, which means that there is a path $e \to v$ in G that does not contain w. Since (v, w) is a cycle edge, $v \in C(w)$. By definition of $P(w)$, we have $e, w \in P(w)$, and by definition of root vertex, we have $w \not\to e$ in T.

($\Rightarrow$) Suppose now that there exists w and $v \in P(w)$ such that there is no path from w to v in T. So there is a path $e \to v$ in T that does not contain w. Since $v \in P(w)$, there exists $z \in C(w)$ such that there is $v \to z$ in T that does not contain w. Combining both paths, there exists $e \to z$ in T, and, so, also in G, that does not contain w, which means that w does not dominate z, but (z, w) is a cycle edge, which is a contradiction with Theorem 1. □

Consider the set of vertices with entering cycle edges. In the following three lemmas, w represents the element of that set with the highest value of $ord(w)$. Suppose that, for all $v \in P(w)$, there is a path $w \to v$ in T. This condition means that there are no jumps to the middle of the loop headed at w. Consider the graph G' obtained from G by collapsing all vertices of $P(w)$ into w.

Lemma 4. *Every edge (v', w') of the graph G' corresponds to an edge (v, w') of the graph G where v is such that there exists $v' \to v$ in T.*

Proof. If (x', y') is an edge of G', either (x', y') was an edge of G, in which case the result trivially holds, or it is the result of a contraction of a vertex of $P(w)$ into w, so either $x' = w$ or $y' = w$.

If $y' = w$, then the edge (x', w) of G' is the result of the contraction of (x, y) of G where $y \in P(w)$, so there exists $z \in C(w)$ such that there is $y \to z$ that does not contain w. Then $x \in P(w)$ combining (x, y) with $y \to z$, so x was contracted to w and $x' = w$. $(x', y') = (w, w)$ is not a proper edge of G'.

If $x' = w$, then the edge (w, y') of G' is the result of the contraction of (x, y) of G where $y \notin P(w)$: if $y \in P(w)$ then $y' = w$ and the edge (w, y') would not be considered. So $y' = y$. Since x was contracted to w, $x \in P(w)$ and, by the previous assumption, we have $w \to x$ in T. □

Let T' be the subgraph of G' whose vertices are those of G' and whose edges are corresponding to the edges of T.

Lemma 5. *(T',e) is a DFST of G'. Cycle edges of G' correspond to cycle edges of G, forward edges of G' correspond to forward edges or cross edges of G, and cross edges of G' correspond to cross edges of G.*

Proof. We start by noticing that e was not collapsed since it has no ingoing edges. T' is a tree: the correspondence between edges ensures that there is one and only one path from e to any vertex $v \neq e$ of G'.

Let (v', w') be a cycle edge of G'. Then there is $w' \to v'$ in T'. Since $T' \subseteq T$, it is also a path of T. By Lemma 4, (v', w') corresponds to an edge (v, w') of G such that there is $v' \to v$ in T. Combining both paths, we conclude that there exists a path $w' \to v$ in T, so (v, w') is a cycle edge.

Let (v', w') be a forward edge of G': $v' \to w'$ is in T'. By Lemma 4 this edge corresponds to an edge (v, w') of G' such that $v' \to v$ is in T. We want to show that (v, w') is a forward edge or a cross edge or, equivalently, that it is not a cycle edge. Assume by contradiction that it is: there exists a path $w' \to v$ in T. Then we can combine these paths: $v' \to w' \to v$. If $v' = v$ this situation is impossible in T. If $v' \neq v$, the edge (v, w') was replaced by (v', w') during the construction of G', which means that $v' = w$, by the proof of the previous lemma. Therefore w' is a vertex with an entering cycle edge that is visited after w in the traversal of T, so $ord(w) < ord(w')$, which contradicts the definition of w.

Let (v', w') be a cross edge of G': $w' \not\to v'$ in T', $v' \not\to w'$ in T' and $w' < v'$. Let v be the source of the edge given by Lemma 4 and recall that $v' \to v$ is in T. Suppose that there exists $v \to w'$ in T. Then there is also $v' \to w'$ in T, which is a contradiction. Suppose now that there exists $w' \to v$ in T. Then, since v has indegree 1, either w' is an ancestor of v' or v' is an ancestor of w', that is, either $w' \to v'$ is in T or $v' \to w'$ is in T, but both options are impossible. □

Lemma 6. *For any vertex x such that $ord(x) < ord(w)$, let $P'(x)$ and $C'(x)$ be defined in G' relative to T' as $P(x)$ and $C(x)$ were defined in G relative to T. Then $x \to y$ in T' for all $y \in P'(x)$ if and only if $x \to y$ in T for all $y \in P(x)$.*

Proof. ($\Leftarrow$) Assume by contrapositive that there exist $x, y \in V$ such that $y \in P'(x)$ and $x \not\to y$ in T'. Then it also holds that $x \not\to y$ in T. Since $y \in P'(x)$, there exists $z' \in C'(x)$ such that there is $y \to z'$ in G' not containing x. By Lemma 4 and Lemma 5, the cycle edge (z', x) of G' corresponds to a cycle edge (z, x) of G. Moreover, there exists a path $y \to z$ in G containing edges from the path $y \to z'$ and possibly some deleted edges with vertices in $P(w) \cup \{w\}$. We are assuming that there is $w \to v$ in T for every $v \in P(w)$, so by Lemma 2 this path cannot contain x since $ord(x) < ord(w)$. So $y \in P(x)$.

($\Rightarrow$) Assume again by contrapositive that there exist $x, y \in V$ such that $y \in P(x)$ and $x \not\to y$ in T.

- If $y \notin P(w)$, then $y \in P'(x)$ and $x \not\to y$ in T';

- If $y \in P(w)$, then it was collapsed to w and, recalling the correspondence between the edges before and after the transformation of the graph, $x \not\to w$ in T'. Since $y \in P(x)$, $w \in P'(x)$.

□

Let us now return to the algorithm. Every vertex v of G is analysed once inside the for loop in lines 6–27. Notice that if v is not the target of a cycle edge, the algorithm skips v and starts analysing the next vertex, since Q is empty when the while loop starts at line 12. The proof of this proposition is only sketched in [17]; herein we present the entire proof.

If w is the vertex of G with entering cycle edges with the highest value of *ord*, then a *contraction* of G is the operation that consists of collapsing all vertices of $P(w)$ into w. Let $G^{(0)} = G$ and $G^{(k)}$ be the result of applying a contraction to $G^{(k-1)}$. Also, let $T^{(k)}$ the DFST of $G^{(k)}$ and $P^{(k)}(w)$, for any vertex w of $G^{(k)}$, be defined relative to $G^{(k)}, T^{(k)}$ as $P(w)$ was defined relative to G, T. Similarly to what we did previously, we may write G' instead of $G^{(1)}$.

Proposition 2. *Algorithm 2 returns* ***true*** *if and only if G is reducible.*

Proof. Let $\{w_1, \ldots, w_l\}$ be the set of vertices of G with entering cycle edges, numbered such that $ord(w_1) > \cdots > ord(w_l)$.

($\Leftarrow$) We will prove that, if G is reducible, then the algorithm returns **true** by induction on the number of vertices with entering cycle edges, l.

Base: If G does not have cycle edges, then it is a tree and it is reducible by repeated application of T_2: every vertex has indegree 1 so, starting from the leaves, it is always possible to collapse a leaf into its parent (the only modification is the removal of one vertex and one edge) until the resulting graph is only the entry node. The algorithm, when applied to a tree, scans every vertex without modifying the tree, eventually reaching line 28 where it returns **true**.

Step: We start by observing that the set of vertices of $G^{(k)}$ with entering cycle edges is $\{w_i \colon i \geq k+1\}$, for all $k \leq l$. By Lemma 4 and Lemma 5, which are applicable since G is reducible, every cycle edge of $G^{(k)}$ corresponds to a cycle edge of G, so the set of vertices with entering cycle edges of $G^{(k)}$ corresponds to the set of vertices with entering cycle edges of G. $find(w_i)$ for $i \leq k+1$ does not have entering cycle edges since they were all collapsed in the previous steps. Finally, w_j is a vertex of $G^{(k)}$ for all $j \geq k+1$, that is, $find(w_j) = w_j$: assume that w_j was previously collapsed; then $w_j \in P^{(i-1)}(w_i)$ for some $i \leq k+1$, so there is $w_i \to w_j$ in $T^{(i)}$ and also in T since w_i and w_j were not already collapsed, but this is a contradiction with $ord(w_i) > ord(w_j)$.

We will show that, if G is reducible, then G' is reducible. By Lemma 3 $\forall x \in V\ \forall v \in P(x)$, there is $x \to v$ in T. By Lemma 6, $\forall x \in V \colon ord(x) < ord(w_1)\ \forall v \in P'(x)$ there is $x \to v$ in T'. Since every vertex x of G' with entering cycle edges is such that $ord(x) < ord(w_1)$, $\forall x \in V'\ \forall v \in P'(x)$ there is $x \to v$ and G' is reducible.

Let us now assume that the first $k < l$ contractions were already made, so the current graph is $G^{(k)}$. We will show that the next iteration of the for loop that changes the graph ($w = w_{k+1}$) transforms $G^{(k)}$ in $G^{(k+1)}$.

During the iteration where $w = w_{k+1}$, $Q \subseteq P \subseteq P^{(k)}(w_{k+1})$: The first inclusion is trivial. For the second inclusion, notice that (u, w) is a cycle edge of G if and only if $(find(u), w)$ is a cycle edge of $G^{(k)}$, by Lemma 4 and Lemma 5; for every $x \in V$, $find(x) \to x$ in G; (y, x) is a forward, tree or cross edge if and only if $(find(y), x)$ is a forward, tree or cross edge; and $w_{k+1} \notin P$. A vertex $z' = find(z)$ is added to P if (z', w_{k+1}) is a cycle edge or if (z', x), for some $x \in Q \subseteq P$, is an edge of $G^{(k)}$. Thus, for every $z \in P$ there exists some u' such that (u', w_{k+1}) is a cycle edge of $G^{(k)}$ there exists a path $z \to u'$ in $G^{(k)}$ that does not contain w_{k+1}.

After the while loop, $P = P^{(k)}(w_{k+1})$: If $z \in P^{(k)}(w_{k+1})$ then there exists u such that (u, w_{k+1}) is a cycle edge in $G^{(k)}$ and there is $z \to u$ in $G^{(k)}$ that does not contain w_{k+1}. $u \in P$ by construction of P. Line 14 builds, backwards, every path of $G^{(k)}$ ending in u that does not contain w_{k+1}. Notice that in line 14 x has no entering edges in $G^{(k)}$ since $ord(x) > ord(w_{k+1})$ was verified before adding x to P. In particular, z is added to P at some point in the execution, so, in the end of the while loop, $z \in P$.

So the last for loop, in lines 24–26, collapses every vertex of $P(w_{k+1})$ into w_{k+1} and the result is $G^{(k+1)}$.

We conclude that, if G is a reducible graph with l vertices, the algorithm successively transforms G in reducible graphs, reducing the number of vertices with entering cycle edges by 1 in each iteration, until $G^{(l)}$ is a tree, and the program returns **true**.

($\Rightarrow$) Assume that G is an irreducible graph. Then there exists k such that $\exists v \in P(w_k)$: $w_k \not\to v$ in T. Since, for all v, there is $find(v) \to v$ in T, this means that $w_k \not\to find(v)$ in T, considering the function $find$ applied to any of the graphs $G^{(j)}$, $j \leq l$. By Lemma 2, $ord(w_k) > ord(find(v))$ or $ord(w_k) + ND(w_k) \leq ord(find(v))$. In iteration k, P eventually analyses all elements of $P^{(k-1)}(w_k)$, by the results obtained in the proof of the previous implication, so it will scan v and return **false**.

□

Complexity analysis

Although Tarjan stated in his 1974 paper ([17]) that the time complexity of this algorithm was $O(|E| \log^* |E|)$, where $\log^*$ is the iterated logarithm, this bound has been later reduced due to results that the same author obtained for the complexity of the operations of the disjoint-set data structure. We will now show that, for n elements, a sequence of m *find*, *union*, and *makeset* operations of which n are *makeset* has worst-case running time $O(m\ \alpha(n))$, where α is related to the inverse Ackermann function.

The Ackermann function was defined in 1928 by William Ackermann ([1]) and it was the first example of a computable function that is not primitive recursive. Since then, several functions that share the same property have been constructed for various purposes. These functions grow faster than any multiple exponential, which means that their inverses grow very slowly – for every practical input, they are always below 5. Our proof and, so, our version of the Ackermann function, follows [4].

We will use amortized analysis to prove the desired time bound. Amortized analysis was first used in the 1980s to prove upper bounds on the time complexity of several algorithms, and it was described in more detail in 1985 by Tarjan in [18]. It is a technique that allows us to obtain tighter bounds

on the execution time of an algorithm by noticing that the worst-case running time of a sequence of operations may be smaller than the sum of the worst-case running times of each operation. One of the ways to perform amortized analysis is to use a potential function. If $\mathcal{D}$ represents all possible states of the data structure used by the algorithm during the execution, then any $\Phi\colon \mathcal{D} \to \mathbb{N}$ is a potential function. The choice of Φ, for each algorithm, strongly depends on the data structure and on the operations that are performed. Having chosen Φ, the amortized time of an operation, relative to Φ, is defined as $a = t + \Phi(D') - \Phi(D)$, where t is the actual time of the operation and $D, D' \in \mathcal{D}$ are the data structure states before and after the operation, respectively. Therefore, a sequence of k operations has amortized time $\sum_{i=1}^{k} a_i = \sum_{i=1}^{k} t_i + \Phi(D_k) - \Phi(D_0)$, so actual time $\sum_{i=1}^{k} t_i = \sum_{i=1}^{k} a_i + \Phi(D_0) - \Phi(D_k)$, where D_0 and D_k are the initial and the final states of the data structure and a_i, t_i are the amortized and actual times of the ith operation. If $\Phi(D_0) < \Phi(D_k)$, then the amortized time is an upper bound on the actual time of the sequence of operations.

As it was already described, the disjoint-set data structure models subsets of a given set. For convenience, since this set will be the set of the vertices of a CFG, we will denote this set by V in the following. Let n be the number of elements of V. We will assume that the first n operations are *makeset* and that the next operations are either *union* or $find$, as it is usual and since it is the case in the algorithm we wish to analyse. Each subset of V is represented as a directed tree, so every vertex has a parent $p(v)$ (the parent of a root is itself). The representative element of each subset is the root of the corresponding tree. Each vertex v has a rank. The operation $makeset(v)$, that creates the tree $\{v\}$, sets $rank(v) = 0$.

Function 1 *makeset*

Input: v

1: $p(v) = v$
2: $rank(v) = 0$

The rank is an upper bound on the height of a node and may be changed by the operation *union* – this variant is usually named *union by rank* and is one of the two heuristics used to achieve the complexity result we are interested in.

Function 2 *union* by rank

Input: u, v

1: $u' = find(u)$
2: $v' = find(v)$
3: **if** $rank(u') > rank(v')$ **then**
4: $\quad p(v') = u'$
5: **else**
6: $\quad p(u') = v'$
7: $\quad$ **if** $rank(u') = rank(v')$ **then**
8: $\quad\quad rank(v') = rank(v') + 1$
9: $\quad$ **end if**
10: **end if**

In the pseudocode above, the roots of the trees corresponding to vertices u and v are merged by making the root with smaller rank a child of the root with higher rank. If the ranks are equal, one of them increases by 1 and the same operation is performed.

The other heuristic is related to the find operation and is called *path compression*. Each time a $find$ operation is executed with input x, not only the root of the tree of x is returned, but also every node on the path from the root to x becomes a direct child or the root. This modification is crucial to guarantee that the heights of the trees are not too big. The $find$ operation does not change any rank, which helps to explain why the rank of a vertex is an upper bound on the height of the corresponding tree. There are several possibilities for path compression; we present a recursive one.

Function 3 $find$ with path compression

Input: v

1: **if** $v \neq p(v)$ **then**
2: $\quad p(v) = find(p(v))$
3: **end if**
4: **return** $p(v)$

This data structure requires $O(n)$ space: it is necessary to store, for each vertex, its parent and its rank. Analysing the time complexity is more complicated and requires some auxiliary functions. We will define a function A_k that is a variant of the Ackermann function.

Definition 19. *Let $k \in \mathbb{N}$. The function $A_k \colon \mathbb{N} \setminus \{0\} \to \mathbb{N}$ is recursively defined as:*

$$A_k(j) = \begin{cases} j+1, & \text{if } k = 0 \\ A_{k-1}^{(j+1)}(j), & \text{if } k \geq 1 \end{cases}$$

where $A_{k-1}^{(j+1)}$ is the function A_{k-1} iteratively applied $j+1$ times.

It is claimed, but not proved, in [4] that the function A_k strictly increases with both j and k. Herein we prove some auxiliary results that will be used later on.

Lemma 7. *For all $k \geq 0$, $j \geq 1$, we have that $A_k(j) \geq j$.*

Proof. We will prove the result by double induction. We start with induction on k.

Base on k: $k = 0$. $A_0(j) = j + 1 \geq j$.

Induction hypothesis on k: For some fixed k and for all $j \geq 1$, $A_k(j) \geq j$.

Induction step on k: For the same k and for all $j \geq 1$, we want to show that $A_{k+1}(j) \geq j$. This will be done using an auxiliary result: for the same k and for all $i, j \geq 1$, $A_k^{(i)}(j) \geq j$. By induction on i:

Base on i: $i = 1$. $A_k^{(1)}(j) = A_k(j) \geq j$ by induction hypothesis on k.

Induction hypothesis on i: For the same k, for some fixed i, and for all $j \geq 1$, $A_k^{(i)}(j) \geq j$.

Induction step on i: $A_k^{(i+1)}(j) = A_k^{(i)}(A_k(j)) \geq A_k(j) \geq j$ where we have used, respectively, the induction hypotheses on i and on k.

Applying this result with $i = j + 1$ it yields $A_k^{(j+1)}(j) \geq j$, which is the same as $A_{k+1}(j) \geq j$, as desired. □

Lemma 8. *For all $k \geq 0$, $j \geq 1$, $A_{k+1}(j) \geq A_k(j)$.*

Proof. $A_{k+1}(j) = A_k^{(j+1)}(j) \geq A_k^{(j)}(j) \geq \cdots \geq A_k(j)$, where we have repeatedly used Lemma 7. □

Lemma 9. *A_k is a strictly increasing function for all $k \geq 0$: if $i < j$ then $A_k(i) < A_k(j)$.*

Proof. Let $i < j$. The proof is by induction on k.

Base: $k = 0$. $A_0(i) < A_0(j) \Leftrightarrow i + 1 < j + 1 \Leftrightarrow i < j$.

Step: The goal is to prove $A_{k+1}(i) < A_{k+1}(j)$, assuming that the same result holds replacing $k+1$ by k. We start by noticing that $i < j$ and $j \leq A_k^{(j-i)}(j)$ by Lemma 7. Then, using the induction hypothesis, the following inequalities are true: $A_k(i) < A_k^{(j-i+1)}(j)$, $\ldots$, $A_k^{(i+1)}(i) < A_k^{(j+1)}(j)$ and the last inequality is, by definition of A_{k+1}, the desired result. □

We define another function, α, that sometimes is referenced as "inverse" of A since it grows as slowly as A grows quickly.

Definition 20. *$\alpha \colon \mathbb{N} \to \mathbb{N}$ is such that $\alpha(n) = \min\{k \colon A_k(1) \geq n\}$.*

The importance of the function α, contrarily to A_k, is mainly practical. For all values of n smaller than the number of atoms in the Universe, estimated to be around 10^{80}, $\alpha(n) \leq 4$, since $A_4(1) > 10^{80}$. Therefore, the running time of a $O(\alpha(n))$ algorithm, despite not being constant, should not vary much according to the input size.

The following properties about *rank* motivate the potential function for this algorithm. Recall that we are considering a disjoint-set data structure over a graph with n vertices.

Lemma 10. *For every node v, it holds that $rank(v) \leq rank(p(v))$, with strict inequality if $v \neq p(v)$.*

Proof. Every vertex v suffers a *union* operation where it becomes a child of another node before suffering a path compression. If $v \neq p(v)$, then v already suffered a *union* where it became the child of a node v' and, in the end of the *union*, $rank(v') > rank(v)$, by definition of union by rank. After that, every operation on v was either *union* – where the relationship between the rank of v and the rank of its parent still holds – or *find* – where v becomes a child of an ancestor r of his and so, inductively, $rank(r) \geq rank(v)$. □

Lemma 11. *For every node v, $rank(v) \leq n - 1$.*

Proof. It is enough to note that the initial rank is 0 and that the rank only increases during the *union* operation, by 1 unit. □

The choice of a good potential function is not straightforward in this context and requires some more definitions. $level \colon V \to \mathbb{N}$ is such that

$$level(v) = \max\{k \colon rank(p(v)) \geq A_k(rank(v))\}$$

Lemma 12. *Let v be a node such that v is not a root and $rank(v) \geq 1$. Then $0 \leq level(v) < \alpha(n)$.*

Proof. Since $v \neq p(v)$, $rank(p(v)) \leq rank(v) + 1 = A_0(rank(v))$, so $level(v) \geq 0$.

$A_{\alpha(n)}(rank(v)) \geq A_{\alpha(n)}(1) \geq n > rank(p(v))$, using Lemma 10, the definition of α, and Lemma 9. So $level(v) < \alpha(n)$. □

Another auxiliary function is $iter\colon V \to \mathbb{N}$:

$$iter(v) = \max\{i\colon rank(p(v)) \geq A^{(i)}_{level(v)}(rank(v))\}$$

iter is the maximum number of times that we can apply $A_{level(v)}$ to $rank(v)$ before this value is greater than $rank(p(v))$.

Lemma 13. *Let v be a node such that v is not a root and $rank(v) \geq 1$. Then $1 \leq iter(v) \leq rank(v)$.*

Proof. By definition of *level*, $iter(v) \geq 1$ for every $v \in V$.

$$A^{(rank(v)+1)}_{level(v)}(rank(v)) = A_{level(v)+1}(rank(v)) > rank(p(v)),$$

by definition of A_k and *level*. □

Let $q \in \mathbb{N}$. We may now define the potential function after q operations on a graph with n vertices, $\phi_q\colon V \to \mathbb{N}$, such that

$$\phi_q(v) = \begin{cases} \alpha(n) \cdot rank(v) & \text{if } v \text{ is a root or } rank(v) = 0 \\ (\alpha(n) - level(v)) \cdot rank(v) - iter(v) & \text{if } v \text{ is not a root and } rank(v) \geq 1 \end{cases}$$

The potential of the entire data structure is the sum of the potentials of each node: for every number of operations q, $\Phi_q = \sum_{v \in V} \phi_q(v)$.

Lemma 14. *For every $v \in V$, and for every $q \in \mathbb{N}$, $0 \leq \phi_q(v) \leq \alpha(n) \cdot rank(v)$.*

Proof. If v is a root, or if $rank(v) = 0$, the result is trivial. Let us now assume that v is not a root and that $rank(v) \geq 1$. Then

$$\begin{aligned} \phi_q(v) &= (\alpha(n) - level(v)) \cdot rank(v) - iter(v) \\ &\leq (\alpha(n) - level(v)) \cdot rank(v) - 1 \\ &\leq \alpha(n) \cdot rank(v) - 1 \\ &\leq \alpha(n) \cdot rank(v) \end{aligned}$$

It also holds

$$\begin{aligned} \phi_q(v) &= (\alpha(n) - level(v)) \cdot rank(v) - iter(v) \\ &\geq (\alpha(n) - level(v)) \cdot rank(v) - rank(v) \\ &> (\alpha(n) - \alpha(n)) \cdot rank(v) - rank(v) \\ &= 0 \end{aligned}$$

□

The next lemma establishes that the potential of a node never increases when an operation is performed and states conditions under which the potential is reduced.

Lemma 15. *Let v be a node that is not a root. Suppose that the qth operation is union or $find$. Then, after the qth operation, $\phi_q(v) \leq \phi_{q-1}(v)$. Moreover, if $rank(v) \geq 1$ and $level(v)$ or $iter(v)$ change due to the qth operation, then $\phi_q(v) < \phi_{q-1}(v)$.*

Proof. Since v is not a root, the qth operation does not change $rank(v)$. The value of $\alpha(n)$ does not change with any operation. Hence, the possible modifications in the potential of v arise from changes to the values of $level(v)$ and $iter(v)$: if $level_k(v)$, $rank_k(v)$ represent the level and the rank of node v after the kth operation, then

$$\phi_{q-1}(v) - \phi_q(v) = rank(v) \cdot (level_q(v) - level_{q-1}(v)) + iter_q(v) - iter_{q-1}(v)$$

If $rank(v) = 0$, then $\phi_q(v) = \phi_{q-1}(v) = 0$.
If $rank(v) \geq 1$, we consider two cases.

- $level(v)$ did not change. Then

  $$\phi_{q-1}(v) - \phi_q(v) = iter_q(v) - iter_{q-1}(v)$$

  $rank(p(v))$ remained unchanged or increased, and using Lemma 7 one concludes that the maximum value of i such that $A^{(i)}_{level(v)}$ is not greater than $rank(p(v))$ cannot be smaller than it was before the qth operation. So $iter(v)$ did not decrease. If $iter(v)$ did not change as well, then $\phi_q(v) = \phi_{q-1}(v)$. Otherwise $iter(v)$ increased, so $\phi_q(v) < \phi_{q-1}(v)$.

- Again, since $rank(p(v))$ either remained unchanged or increased, and by Lemma 8, if $level(v)$ changed then it increased, so

  $$\phi_{q-1}(v) - \phi_q(v) > rank(v) + iter_q(v) - iter_{q-1}(v)$$

  By Lemma 13, $iter_q(v) - iter_{q-1}(v) \geq 1 - rank(v)$, so $\phi_q(v) < \phi_{q-1}(v)$.

□

We are now ready to compute the amortized cost of each operation.

Proposition 3. *The amortized costs of each disjoint-set data structure operation over a structure with n vertices are the following:*

1. *makeset:* $O(1)$;
2. *$find$:* $O(\alpha(n))$;
3. *union:* $O(\alpha(n))$.

Proof. Recall that the amortized cost of an operation is its actual cost plus the change of potential due to the operation.

1. This operation adds a new node with rank 0 and does not change any existing ranks, so it does not change any *level* or *iter* and $\Phi_q = \Phi_{q-1}$. Since the actual cost of the operation is $O(1)$, the amortized cost is also $O(1)$.

2. The actual cost of $find(v)$ is $O(s)$, where s is the number of nodes in the path between the root of the tree where v is and v. In the worst case $s = n$ but, since that cannot be the case in every $find$ call contained in a sequence of operations, the amortized cost will be less than $O(n)$.

   $find$ does not increase the potential of any node: if v is not a root, by Lemma 15 we have that $\phi_q(v) \leq \phi_{q-1}(v)$; if v is a root then $\phi_{q-1}(v) = \alpha(n) \cdot rank(v) = \phi_q(v)$ since the $find$ operation does not change any rank.

   $find$ decreases the potential of at least $\max(0, s - \alpha(n) - 2)$ nodes: we will define a set of nodes that has at least $s - \alpha(n) - 2$ elements and then show that the potential of each element decreases. Consider the $find$ operation applied to node v. Let r be the root of the tree of v.

   $$X = \{x \colon x \to v, rank(x) > 0, \exists y \colon y \neq r, y \to x, level(x) = level(y) \text{ before } find\}$$

   Let p be the path $r \to v$. X is the set of elements in p whose ranks are not 0 and such that there exists an ancestor y of x, also in p, such that the levels of y and x are equal. Recall that *level* is (non-strictly) decreasing in p and that there are at most $\alpha(n)$ levels. *level* is not defined for r, so $r \notin X$. The first node of p to have a certain level k, for each k, is also not in X. If v has rank 0, we also have $v \in X$. All remaining nodes of p should be in X since their ranks are all positive and we already excluded those that do not satisfy the condition on *level*. So, X has at least $s - \alpha(n) - 2$ elements. Let $k = level(x) = level(y)$ and $i = iter(x)$ before the $find$ operation. Then, using the definitions of *level*, *iter*, $rank(y) \geq rank(p(x))$ and the fact that A_k is increasing,

   $$rank(p(y)) \geq A_k(rank(y)) \geq A_k(rank(p(x))) \geq A_k(A_k^{(i)}(rank(x))) = A_k^{(i+1)}(rank(x))$$

   After $find$, $p(y) = p(x)$, $rank(x)$ is the same and $rank(p(y))$ did not decrease. Therefore $rank(p(x)) \geq A_k^{(i+1)}(rank(x))$. By definition of *iter*, if *level* did not change, then *iter* increased, and by Lemma 15 $\phi_q(x) < \phi_{q-1}(x)$. Otherwise *level* changed and we obtain the result by the same Lemma. Thus, $\Phi_q - \Phi_{q-1} \geq -(s - \alpha(n) - 2)$ and the amortized cost of the operation is at most $O(s) - (s - \alpha(n) - 2) = O(\alpha(n))$ scaling up the units of potential.

3. A *union* operation consists of two $find$ calls and some operations whose actual time is constant but that may change the potential of the data structure. There are three reasons that may change the potential of a node v: $rank(v)$ changed; $p(v)$ changed, $rank(p(v))$ changed. Assume that the operation *union* receives nodes x and y and that x becomes a child of y.

- Change of rank: the only node whose rank may change is y. $\phi_{q-1}(y) = \alpha(n) \cdot rank(y)$. Either $\phi_q(y) = \phi_{q-1}(y)$ or $\phi_q(y) = \phi_{q-1}(v) + \alpha(n)$;
- Change of parent: the only node whose parent may change is x. If $rank(x) = 0$, then $\phi_{q-1}(x) = \phi_q(x) = 0$. Otherwise, since x is not a root anymore, $\phi_q(x) = (\alpha(n) - level(x)) \cdot rank(x) - iter(x) < \alpha(n) \cdot rank(x) = \phi_{q-1}(x)$.
- Change of rank of parent: the only nodes in these conditions are the children of y before *union*. Their potentials do not change due to Lemma 15.

Hence, $\Phi_q \leq \Phi_{q-1} + \alpha(n)$. The amortized cost is, at most, using the previous result for $find$, $O(\alpha(n)) + O(1) + \alpha(n) = O(\alpha(n))$.

□

The main result follows easily.

Proposition 4. *A sequence of m makeset, union, and $find$ operations, n of which are makeset, can be performed on a disjoint-set data structure in worst-case time $O(m\alpha(n))$.*

We are finally ready to analyse the time and space complexity of Tarjan's algorithm (2).

Proposition 5. *Tarjan's algorithm, when given an input graph G with n vertices and m edges, has worst-case running time $O((n+m)\alpha(n))$ and requires $O(n+m)$ space.*

Proof. The construction of a DFST of G, in line 1, is known to take $O(n+m)$ time and space. Recall that DFS maintains a structure to record the vertices that were already visited. The function *ord* is calculated during the traversal in constant time for each vertex. ND can be calculated, for each vertex v, during the construction of the tree: when v is visited for the first time, $ND(v)$ is set to 0; every time that an already visited vertex v is revisited, $ND(v)$ is incremented by 1. Each of these operations requires constant time and they are executed in every visit to a vertex, so the asymptotic time complexity does not change. The additional space is $O(n)$, so the space also remains unchanged.

We can assume that G is stored using adjacency lists: for each $v \in V$, the adjacency list of v contains every edge ending in v. The first for loop, in lines 2–5, will scan every edge in the adjacency list of v in order to determine if it is a cycle edge, a forward edge or a cross edge. Recall that if there is $v \to u$ in T then (u, v) is a cycle edge, if there is $u \to v$ in T then (u, v) is a forward edge and otherwise (u, v) is a cross edge. This verification can be done using Lemma 2 in constant time, given that all values of n and ND were previously stored. Thus, this loop needs $O(n+m)$ time and at most $O(m)$ additional space.

The main for loop, in lines 6–27, requires a more delicate analysis. Since we are interested in worst-case behaviour, we assume that the loop runs once for each vertex without ever returning **false**. We start by noticing that the loop in lines 8–10 runs exactly once for each cycle edge of G and that the inner cost is the cost of the operation $find$. Every vertex of G is in P at most once during the algorithm: once it is in P, it is collapsed into other vertex w that was not in P yet, by the proof of correction of the algorithm. If the graph is reducible then the procedure ends with a graph that only has one vertex, so every vertex but one was collapsed, which means that every vertex but one was in P. Therefore every forward or

cross edge of G is considered exactly once during all executions of the for loop in lines 14–22. The cost of executing the body of the loop is of the order of the cost of $find$, since the remaining operations need a constant number of steps – checking if y is a member of P can be done in constant time if P is implemented as an array containing 1 in position y if $y \in P$ and 0 otherwise; Q can be implemented as a queue or a stack so that the selection of a vertex in line 13 takes constant time; copying all elements of P to Q in line 11 takes, during the whole algorithm, $O(n)$ time since every vertex is in P at most once. The for loop in lines 24–26 runs a total of $n-1$ times. Hence, the total time required for this loop is of the order of $O(n+m)$ plus the time required to perform m $find$ operations plus the time required to perform $n-1$ *union* operations. P and Q require at most $O(n)$ space each.

The total space needed for this algorithm is bounded by $O(n+m)$. The total time of executing n *makeset*, m $find$, and $n-1$ *union* operations is, by 59 4, $O((n-1+m)\alpha(n)) = O((n+m)\alpha(n))$. The total time of the algorithm is $O(n+m) + O((n+m)\alpha(n)) = O((n+m)\alpha(n))$. □

4.2.2 Finding natural loops

We have claimed before that reducible control flow graphs are important because they correspond to programs that only have natural loops. Such a loop is always associated with a back edge. Recall that a back edge is an edge such that its target dominates its source.

Definition 21 (Natural loop; header)**.** *Let $G = (V, E)$ be a CFG. A natural loop of a back edge $(u, w) \in E$ is the subgraph $G_{loop} = (V_{loop}, E_{loop})$ of G such that $V_{loop} = \{w\} \cup \{v \colon \exists\ v \to u$ that does not contain $w\}$ and $E_{loop} = \{(u, v) \in E \colon u, v \in V_{loop}\}$. w is the header of the natural loop.*

In the following, for simplicity, we may use the expression *natural loop of a CFG* to refer to a *natural loop of a back edge of the CFG.*

Since, in reducible graphs, back edges and cycle edges are the same, we may state informally that, in a reducible graph, every loop is a natural loop.

If G is reducible and if the back edge, or cycle edge, (u, w) is the only cycle edge entering w, the natural loop of (u, w) is the subgraph of G induced by $P(w) \cup \{w\}$, that is, the graph $G_{loop} = (V_{loop}, E_{loop})$ such that $V_{loop} = P(w) \cup \{w\}$ and $E_{loop} = \{(u, v) \in E \colon u, v \in V_{loop}\}$.

The following result justifies why the analysis of natural loops is much simpler than the general analysis of loops.

Lemma 16. *Let G be a CFG with entry node e. Two natural loops of G with distinct headers are either disjoint or nested; that is, if $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are the natural loops of the back edges (u_1, w_1), (u_2, w_2) of G, with $w_1 \neq w_2$, then one of the following three options holds: $V_1 \subseteq V_2$, $V_2 \subseteq V_1$, $V_1 \cap V_2 = \emptyset$.*

Proof. Suppose that the loops are not disjoint: $V_1 \cap V_2 \neq \emptyset$.

Assume that $w_1 \in V_1 \cap V_2$. Then there exists $w_1 \to u_2$ without w_2. Since (u_1, w_1) is an edge of G, there exists $u_1 \to u_2$ without w_2, so $u_1 \in V_2$. Analogously, if $w_2 \in V_1 \cap V_2$, then $u_2 \in V_1$. Therefore, if $V_1 \cap V_2 \neq \emptyset$, then $V_1 \cap V_2 \setminus \{w_1, w_2\} \neq \emptyset$.

Let $v \in V_1 \cap V_2 \setminus \{w_1, w_2\}$. Then there are paths $v \to u_1$, $v \to u_2$ without w_1, w_2, respectively. Since w_1 dominates u_1 and w_2 dominates u_2, every path $e \to u_1$, $e \to u_2$ contains w_1, w_2, respectively. Then, every path $e \to v$ contains both w_1 and w_2, so there exists $w_1 \to v$ without w_2 or $w_2 \to v$ without w_1. Without loss of generality assume $w_2 \to v$ without w_1. We will show that $V_2 \subseteq V_1$:

- $w_2 \in V_1$: combining the paths $w_2 \to v$ and $v \to u_1$ results in $w_2 \to u_1$. None of the original paths contains w_1, so the combined path also does not contain w_1.
- Let $z \in V_2 \setminus \{w_2\}$. $z \in V_1$: there exists $z \to u_2$ and (u_2, w_2) is an edge of G, so there exists $z \to w_2$. Then, combining this path with the path that ensures that $w_2 \in V_1$, that does not contain w_1, we conclude that there exists $z \to u_1$ that does not contain w_1.

If we had assumed that there was $w_1 \to v$ without w_2 we would have proved analogously that $V_1 \subseteq V_2$. □

Notice that two natural loops may share the same header and have distinct bodies, but if the header of a loop $G_1 = (V_1, E_1)$ belongs to another loop $G_2 = (V_2, E_2)$ and it is not the header of G_2, then $V_1 \subseteq V_2$, that is, the entire loop G_1 is nested in G_2.

Definition 22 (Simple natural loop; inner loop; maximal inner loop)**.** *Let $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$ be natural loops of G.*

- *G_1 is simple if it does not contain any other natural loop;*
- *If $V_2 \subsetneq V_1$ we say that G_2 is an inner loop of G_1;*
- *If there is no natural loop $G_3 = (V_3, E_3)$ such that $V_2 \subsetneq V_3 \subsetneq V_1$, we say that G_2 is a maximal inner loop of G_1.*

A natural loop may have several maximal inner loops, but in that case they are disjoint.

If a graph G with n vertices and m edges is reducible, Tarjan's algorithm computes all natural loops, but collapses them before starting the next iteration. Moreover, if there are l natural loops, a complete list of them may require $O(nl)$ space: vertices that are common to several nested loops need to be stored in each individual loop. The time complexity would be at least $O(nl)$, plus $O(n + m)$ for the analysis of every vertex and edge. For this reason, we will develop an algorithm that saves us some space and time. Since the input G is reducible, we use the terms *back edge* and *cycle edge* interchangeably.

Some observations regarding the differences between both algorithms:

- The disjoint-set data structure is not used anymore: since we need to return vertices of the original graph, we chose not to destroy its structure;
- We have removed the verification of the existence of a path between y and w, in lines 19–21 of Algorithm 2, because the input G of this algorithm is reducible;
- For each w, C is the set of sources of cycle edges entering w and it is introduced in order to build the natural loop of every cycle edge entering w separately;

Algorithm 3 Algorithm to find implicit natural loops based on Tarjan's algorithm

Input: reducible control flow graph $G = (V, E)$ such that $|V| = n$

Output: O

construct a DFST of G numbering the vertices from 1 to n and calculating $ND(v)$ for each vertex v
for v such that $ord(v) = 1$ until $ord(v) = n$ **do**
    make lists of cycle edges, forward edges and cross edges that enter v
    $U[ord(v)] = 0$
end for
$O = \emptyset$
for w such that $ord(w) = n$ **step** -1 until $ord(w) = 1$ **do**
    $P = \emptyset$
    $C = \emptyset$
    for each cycle edge (u, w) **do**
        add u to C
        $U[ord(u)] = w$
    end for
    while $C \neq \emptyset$ **do**
        select a vertex c from C and add it to P and to Q
        delete c from C
        while $Q \neq \emptyset$ **do**
            select a vertex x from Q and delete it from Q
            for each forward edge or cross edge (y, x) **do**
                if $U[ord(y)] \neq 0$ **then**
                    add $U[ord(y)]$ to P and to Q
                else if $y \notin P$ and $y \neq w$ **then**
                    add y to P and to Q
                    $U[ord(y)] = w$
                end if
            end for
        end while
        $U[ord(w)] = w$
        add $\{w\} \cup P$ to O
        $P = \emptyset$
    end while
end for
return O

- U contains, for each vertex v, the header of the smallest loop where v is.

Lemma 17. *For each natural loop $L = (V_L, E_L)$ of G, let $L_1 = (V_1, E_1), \ldots, L_k = (V_k, E_k)$ be the maximal inner loops of L, with headers $w_1, \ldots, w_k$, respectively. The output O of Algorithm 3 contains, for each L, a set $A_L \subseteq V_L$ such that $\{w_1, \ldots, w_k\} \cup (V_L \setminus (V_1 \cup \cdots \cup V_k)) \subseteq A_L$, and every element of O is of this form for some natural loop L.*

Proof. Let $A \in O$. Then $A = \{w\} \cup P$ for some w with entering cycle edges and some set of vertices P. There exists u such that (u, w) is a cycle edge. Let L be the natural loop of (u, w). $u \in P$ because u was added to C in line 11 and every element of C is eventually added to P in line 15. When the while loop in line 17 starts, P and Q only have one element, u: even if another cycle edge entering w was processed before (u, w), P and Q were cleared before u became an element of P and Q. Then, every element $v \neq u$ in P was added during the while loop, either because it had a forward or cross edge to an element of Q or because it was the header of a natural loop containing a vertex that had a forward or cross edge to an element of Q. There is a path from a header of a loop to any vertex of that loop. Furthermore, if v was such that $U[ord(v)] \neq 0$, then the entire loop was previously visited, so it is an inner loop of the natural loop of (u, w) and the path from the header to v does not contain w. Hence, $v \to u$ without w for all $v \in P$. Then $A \subseteq V_L$.

If in line 20 the condition is true, then y is a member of an inner loop L' of the natural loop of (u, w) and the header of L', $w_{L'}$, is added to P. If L' is not maximal, then $w_{L'}$ is contained in another inner loop of L and it is not its header, so the edges $(y, w_{L'})$ found in line 19 will return vertices y that were already visited and we will add the header of y to P and Q. This process eventually ends with the header of a maximal inner loop of L being added to P. We do not skip any header of a maximal inner loop since they are disjoint and we only skip paths inside inner loops. So $\{w_1, \ldots, w_k\} \subseteq A$.

A contains every vertex of V_L that is not a vertex of any inner loop of L: if we did not skip vertices of inner loops, like in Tarjan's algorithm, the output would contain every vertex in V_L. Vertices that are skipped are part of some inner loop of L, so $V_L \setminus (V_1 \cup \cdots \cup V_k) \subseteq A$.

For each natural loop its cycle edge is considered in line 10 and its source is added to C, so using a reasoning similar to the previous one $\{w_1, \ldots, w_k\} \cup (V_L \setminus (V_1 \cup \cdots \cup V_k)) \subseteq A_L$. □

In particular, if L is a simple natural loop, then A_L is exactly the set of vertices of L. If L is a natural loop with header w_L containing exactly one inner loop, which is simple, we can obtain V_L from A_L by looking for a header in A_L different from w_L. Looking for a header can be done in constant time if we consider an additional structure, like an array, that stores which vertices are headers. In this case there is only one, say w, and we can search the natural loops headed by w in O and copy those vertices to A_L. Then V_L is complete, recalling that A_L already contained the vertices that are exclusive to L. In general, if L is a loop with several nested inner loops, this procedure allows us to construct V_L since the innermost loop of each chain of nested loops is simple and the vertices that are exclusive to each nested loop L' are in $A_{L'}$.

Proposition 6. *Algorithm 3 given an input G with n vertices and m edges, takes, in the worst case,*

$O(n+m+c^2)$ time and $O(n+m+c^2)$ space, where c is the maximum number of natural loops inside a natural loop of G.

Proof. The construction of the DFST, along with the calculation of ord and ND for each vertex, was already shown to need $O(n+m)$ time and space. Splitting the edges entering every vertex in cycle, forward, and cross edges also takes $O(n+m)$ time using ND and $O(m)$ additional space.

Every cycle edge of G is considered exactly once in lines 10–13. If a vertex v is in Q more than once during the execution, then v is a header of some inner loop of the current natural loop. A second visit to a forward or cross edge in line 19 only occurs when $U[ord(y)]$ was added to P and Q instead of y in the previous step and we are now considering an edge whose target is $U[ord(y)]$ – a header that was previously visited. So when a loop has c loops inside it, the total number of repetitions of edges and vertices is at most $2(1+2+\cdots+c) = c(c+1)$. So the total time is $O(n+m+c^2)$.

P, Q, C, and U need $O(n)$ space. O needs $O(n+c^2)$ space considering the repeated headers. The total space is therefore $O(n+m+c^2)$. □

Constant c can be of the order of the number of natural loops of G, previously defined as l. The direct adaptation of Tarjan's algorithm led to a $O(nl+m)$ complexity; this algorithm changes it to $O(n+m+l^2)$, which is smaller when $l < n$, and usually that is the case. It should be noted that c can be higher than the nesting level if there are disjoint loops inside a loop, but it is typically a small number.

Consider the control flow graph of an EVM program. It was already observed that, for every vertex v of the CFG, $deg^+(v) \leq 2$. Then every edge is in at most two circuits (without repeated vertices in each circuit) of the graph, so it is in at most two natural loops of the graph. This means that the maximum number of natural loops of the graph is $2n$, so $c \leq 2n$. We could write the temporal complexity of the previous algorithm as $O(n^2+m)$, or $O(n^2)$. However, in real programs, it is likely that c is much smaller than n, so we chose to state the complexity of the algorithm in terms of c.

Having identified the loops of a program, we return to our original goal of calculating its gas cost. If a simple natural loop runs a constant number of times, x, then the cost of executing the loop is x times the cost of executing the loop body. But the number of iterations of a loop cannot be inferred from the CFG and determining it may not be a simple task.

Lemma 18. *Computing the exact number of iterations of a loop is NP-hard.*

Proof. Consider a boolean formula φ written in conjunctive normal form (CNF) such that every clause of it has 3 literals. Let $x_1, \ldots, x_n$ be the variables of φ. 3-SAT is the problem of determining whether there exists a satisfying assignment for φ and it is known to be NP-complete. The problem of computing a satisfying assignment for φ or returning **false** if there is none is self-reducible to 3-SAT, so it is also NP-complete.

Given a formula φ, consider the program P_φ that computes the number of variables of φ, n, and tries to find a satisfying assignment for φ by trying every $(b_1, \ldots, b_n) \in \mathbb{Z}_2^n$. It stops when $\varphi(b_1, \ldots, b_n)$ is **true**, returning the assignment, or when it runs out of possibilities, returning **false**. The assignments are tested in ascending order: $(0, \ldots, 0, 0), (0, \ldots, 0, 1), \ldots, (1, \ldots, 1, 1)$. It is easy to see that we can write this program in polynomial time in the size of φ.

If P_φ stops after k iterations, then that the kth assignment is a satisfying assignment for φ. Conversely, if $(b_1, \ldots, b_n)$ is a satisfying assignment for φ, then the number $(b_1 \ldots b_n)_2$, interpreted in base 10, is the number of iterations of P_φ.

Therefore, 3-SAT is polynomial-time reducible to our problem, which means that computing the exact number of iterations of a loop is NP-hard. □

In general, the execution of an iteration of a loop is dependent on the truth value of a condition. The meaning of Lemma 18 is that solving that condition for every possible state of the program when it reaches its verification would be equivalent to solving SAT. Of course, there are cases where the number of iterations is easy to determine: a `for` loop where the initial and the final values of the iterator variable, as well as its increment, are constant; or a `while` loop whose stopping condition is simple. Programs with these loops will be the ones for which we can give a precise estimate of their gas cost, as it will be seen in the next section.

4.3 Description of the tool

General description

The tool was designed to deal with bytecode resultant from compiling typical smart contracts written in Solidity. Recall that such contracts contain global variables and functions, and that a function f that receives n arguments of types $t_1, \ldots, t_n$ is identified using the first 4 bytes of the hash of the string "$f(t_1, \ldots, t_n)$" – commonly referred to as the "hash of the function". The gas calculator receives three arguments: a piece of bytecode (the program), a hexadecimal string with 4 bytes (the hash of the function to execute), and the set of hashes of the functions of the program. If the second argument is a member of the third argument, the corresponding function is simulated; if the second argument is the empty string, the fallback function is simulated; otherwise an error is returned. The tool then returns a list of pairs of the form $(cost, condition)$, where *cost* may be a number or an expression depending on input, transaction or global state parameters, and *condition* is a formula with a structure similar to the one described in Section 3.2.1.

It can handle bytecode whose control flow graph does not have circuits and bytecode whose control flow graph has structured circuits, which we will call *structured for loops*. We define *structured for loop* (in terms of both bytecode and its control flow graph) as a subset of the set of basic blocks of a program such that:

- it is a natural loop;
- the number of iterations of the natural loop is constant;
- the header of the natural loop ends with a conditional jump (`JUMPI`) whose condition is the comparison of an execution stack element with a fixed integer through one of the signs $<$, $\leq$, $>$, $\geq$;
- the stack element that is compared with a fixed integer (the iterator variable) is incremented or decremented by the same amount after each iteration of the natural loop;

- the iterator variable is either the content of a position of the storage of the contract (usually, a global variable of the contract) or an element of the execution stack (usually, a local variable of the function);
- the only possible way to exit the loop is through the header.

These requirements are heavily influenced by both theoretical and practical concerns. The choice of natural loops is motivated by the definition of the concept of loop and by the necessity of classifying a basic block as "inside a loop" or "outside a loop" when calculating its contribution to the total execution cost. Since some analysis of the bytecode needs to be done dynamically, when executing a path its loops must be unfolded and, as such, we require the number of iterations of every loop to be fixed. The next three conditions reflect the usual structure of a compiled `for` loop and allow us to compute the exact number of iterations from key values about the iterator variable: the initial value, the final value, and the increment between iterations. The final condition states that there are no feasible exits mid-loop – there are no possible execution paths containing jumps from an element of the loop body to an instruction that is outside of the loop.

Circuits in the CFG may not correspond exactly to the classical loops found in source code, so it is difficult to provide a simple description of the categories of high-level code that are analysed by this tool. It is clear that `for` loops that run through elements of a list with variable size and `while` loops fall under the category of cycles that the tool cannot handle, but there are some other limitations. Solidity automatically generates a getter function for each global variable of a contract. If there is a global variable of the type `string` or `bytes`, or an array whose elements are of any type, its getter method uses a loop to read the string characters that will not be processed by the tool.

Remix is an online Solidity compiler that allows users to deploy code to the Ethereum blockchain and that contains several functionalities related to the analysis of code: it helps to correct syntax errors, warns the user if a gas-expensive pattern is being used, and displays the compiled code in such a way that helps to link a piece of bytecode to the corresponding piece of source code. It also contains a gas estimator that gives, for each function, a numerical value that is an upper estimate on the cost of calling it, or "infinite" if a finite estimate cannot be found. Our tool is an improvement over that of Remix since it provides a set of possible costs along with the situations where they apply. As it will be seen in Section 4.4, the variability in the cost of a function caused by external parameters is very high, so this classification is of great importance. Also, the gas estimator of Remix does not support loops yet, yielding "infinite" whenever a function contains a cycle (explicit or implicit – caused by an array variable), while our tool accepts loops with a fixed number of iterations. Finally, our path-centered approach allows us to rule out mutually exclusive conditions over gas costs, whereas the block-centered approach of Remix may overestimate costs in some cases. On the other hand, one downside of our tool is that we either accept a program and fully analyse it, or reject it and do not return any values, while the estimator of Remix may yield "infinite" for some functions and a concrete value for other functions of the same contract. Our option is, again, related to the path-centric approach that was followed: functions are not isolated entities, they are part of a contract, so in order to simulate all execution paths of a function the full

control flow graph of the contract is needed.

The symbolic virtual machine used to perform this analysis is an adaptation of the one developed for the formal procedure described in Chapter 3. Since the goal is now to determine gas costs and not possible outcomes, some assumptions were strengthened while some were weakened. Stronger assumptions are related to the loop structure described above. The weaker ones regard global state, transaction information, and storage and memory contents. In order to analyse the header of a loop it is important to run this basic block independently of the context so that the condition that determines whether the execution enters the loop can be isolated and interpreted generally. For example, a typical condition is $e_i < x$, where e_i is the ith element of the execution stack before executing the header block and x is a fixed element of $\mathbb{N}_{256}$. Thus, this variation of the symbolic machine is able to start executing bytecode at any point, considering variables for execution stack elements when needed. It can also consider symbols for storage or memory contents if no element was stored there during the present execution. This way it supports both local (part of the execution stack) and global (saved in the storage of the contract) iterator variables. Given that the machine runs with no input, calls to other contracts are only simulated.

Gas costs

The complete list of gas costs of the original opcodes of the EVM can be consulted in [19]; the costs of the recently added opcodes `REVERT`, `RETURNDATACOPY` and `RETURNDATASIZE` are equal to those of the previously existing operations `RETURN`, `CALLDATACOPY` and `CALLDATASIZE`, respectively. The cost of an opcode `OP` whose list of arguments is l is, in general, equal to

$$cost(\texttt{OP}, l) = C_{mem}(i', i) + c^*(\texttt{OP}, l) \tag{4.4}$$

c represents the part of the cost that is not influenced by memory and C_{mem} is a function that returns the part of the cost of an operation due to increase in memory usage[1]:

$$C_{mem}(i', i) = 3(i' - i) + \left\lfloor \frac{i'^2}{512} \right\rfloor - \left\lfloor \frac{i^2}{512} \right\rfloor \tag{4.5}$$

i and i' are the numbers of active words in memory before and after executing the opcode. The vast majoirty of the opcodes does not change the number active words in memory and has a constant gas cost; our concern will be with the opcodes whose cost is variable. Their cost in the context of a particular execution will be computed dynamically in order to capture the most information about the variables that influence them. Then, the number of active words in memory is, in general, known, and so the contribution of the function C_{mem} to the cost is also known. The following list presents the opcodes whose c^* function mentioned previously is not constant, describing the cases that are considered by the tool. We write $\texttt{OP}(i_1, \ldots, i_n)$ if the opcode `OP` takes n elements from the execution stack and $i_1, \ldots, i_n$ are the first n elements of the execution stack when it is executed. Recall from Chapter 2 that $\iota.\texttt{actor}$ is the address of the contract that is currently executing and $\mu.\texttt{gas}$ is the available gas for the current execution.

[1] Our definition of C_{mem} is taken from [6]; the yellow paper [19] defines $C_{mem}(i) = 3i + \left\lfloor \frac{i^2}{512} \right\rfloor$ and considers that the increase in cost is $C_{mem}(i') - C_{mem}(i)$.

- `SSTORE`(p, c) costs 5000 gas if $c = 0$ or if $\sigma(\iota.\texttt{actor})(p) \neq 0$; otherwise it costs 20000 gas. The tool duplicates the number of current call stacks, substituting each one by a copy of it where $c = 0 \vee \sigma(\iota.\texttt{actor})(p) \neq 0$ is added to the current condition of the call stack and 5000 gas is added to the gas spent, and another where $c \neq 0 \wedge \sigma(\iota.\texttt{actor})(p) = 0$ is the condition added and 20000 gas is added to the gas spent;

- `SELFDESTRUCT`(a) also has two possible costs: 5000 gas if the account a exists, and 37000 gas if the account a does not exist. Similarly to the `SSTORE` case, each option is considered in a copy of the current call stack;

- `LOG`$n(p, s)$[2], for each $n = 0, 1, 2, 3, 4$, has a cost of $375 + 375 \cdot 8 \cdot n \cdot s$. While n is known for each opcode, s comes from the execution stack and, so, the expression $375 + 375 \cdot 8 \cdot n \cdot s$ is added to the gas spent;

- `EXP`(b, x) has a cost of $10 + 50(1 + \lfloor \log_{256} x \rfloor)$, which is added to the gas spent;

- `SHA3`(p, s) has a cost of $30 + 6\left\lceil \frac{s}{32} \right\rceil$;

- `CALLDATACOPY`(m, d, s), `RETURNDATACOPY`(m, d, s), `CODECOPY`(m, d, s), and `EXTCODECOPY`(a, m, d, s) have a cost of $3 + 3\left\lceil \frac{s}{32} \right\rceil$;

- `CALL`, `CALLCODE`, and `DELEGATECALL` are instructions to enter the execution of other contract that send, among other input information, some gas to it. Since our analysis aims to be general and does not take concrete input, it does not execute the call, it only simulates it. Therefore, it is not possible to know how much of the gas sent was actually spent in the inner execution, so the value considered is the gas sent. This may be a numerical value or an expression depending on the existence of the account to call and execution stack elements. The cost, excluding C_{mem}, is given by:

$$c^*(\texttt{CALL}, \{g, to, va, io, is, oo, os\}) = C_{base}(va, flag) + C_{gascap}(va, flag, g, \mu.\texttt{gas}) \tag{4.6}$$

  The functions C_{base} and C_{gascap} are defined in Appendix A and $flag$ is a boolean value indicating whether the account whose address is to exists or not. As expected, this function is bounded by $\mu.\texttt{gas}$, but the advantage of considering this approach is that, in some cases, it is possible to obtain a lower upper bound.

Thus, to summarize, we divide the opcodes in three categories: those with a constant cost; those with a small number of possible costs, where each option is taken separately; and those with a cost given by an expression. The final result, if it is a variable expression, may uncover the input variables that can cause the program to have a greater cost than expected or desired, but it may also be formatted to show only numerical upper bounds for better readability.

[2]Each `LOG`n receives, in reality, n+2 arguments; we did not write them explicitly because the remaining n are execution stack elements that do not influence the cost.

Methodology

The tool starts by obtaining the control flow graph of the program, which is fundamental to determine the possible execution paths. It does so through a procedure that combines static and dynamic information. In an initial phase, the basic blocks are computed and the static part of the control flow graph is built: every basic block is scanned and the jumps (conditional and unconditional) that are both static and valid are added to the graph. Blocks that end with dynamic jumps form the set V' of the blocks that need further analysis. The next phase consists of iterating a two-step procedure until the resulting graph no longer changes:

1. Execute every path starting in the entry node and ending in a block $v \in V'$ except the last instruction (the unknown jump). The element on top of the stack, if it is a valid destination, is a possible destination of the block v. Add the corresponding edge to the graph. If the jump is conditional, also add the edge connecting the current block to the next block to the graph. Notice that other execution path ending in v might end in a different destination, which is why vertices are never removed from V'. Iterate this procedure until a fixed point is reached;

2. Run Tarjan's algorithm on the weakly connected component of the current graph that contains the entry node. This determines whether the graph is irreducible or not while computing all cycle edges of the analysed portion of the program. If it is irreducible, stop and return an error message – it is not possible to estimate gas costs for an unstructured program. Otherwise, an auxiliary procedure that characterizes the natural loop associated to each cycle edge is run – a cycle is then a tuple $(init, guard, body, final, iter, exits)$, where *init* is the initialisation block, where the iterative variable is set to its initial value; *guard* is the block containing a conditional jump such that one of the possibilities enters the loop; *body* is the set of execution paths starting in the destination of the jump of *guard* that goes inside the loop and ending in *guard*; $final$ is the other destination of *guard*; *iter* is the number of iterations of the cycle; and *exits* is the set of edges of the current graph that start in a vertex of *body* (except *guard*) and end in a vertex outside *body*. If more than one initialisation block is detected, or no final block is detected, or the condition is not of the previously specified form, or *iter* is not constant, or at least one execution path containing an edge of *exits* is possible, the execution stops and an error message is returned. Otherwise, the paths containing the natural loops of the program are executed, repeated the necessary number of times, and the destinations of the dynamic jumps that follow those loops are determined, which updates the graph.

The control flow graph returned in the end is the weakly connected component of the result graph that contains the entry node.

Then, in order to compute the gas cost of executing a given function, it is necessary to identify the subset of the CFG that corresponds to that function and run every execution path of it, including executing cycles the correct number of times. A callstack is initialized with a condition indicating that the first part of the input is equal to the specified hash, if it is not empty, and a condition indicating that it is different from every function hash of the program, if it is empty (fallback function). Paths taken

from the CFG are simple (do not enter loops), so the execution of a path is preceded by a preprocessing function that reads the path, compares its content with the determined cycles, and unfolds the loops contained in the path. If the *body* of a cycle contains more than one execution path, all sequences of *iter* paths from *body* are considered. Paths that are determined to be impossible at runtime are discarded and do not influence the possible costs.

The resulting list of call stacks is then parsed to obtain a simpler list of pairs $(cost, condition)$, one for each call stack, such that *cost* is the initial gas minus the parameter $\mu.\texttt{gas}$ and *condition* is $\mu.\texttt{cond}$ of the top of the call stack. The final output is the result of simplifying this list by combining pairs with equal costs in a single pair such that the first entry is the common cost and the second entry is the disjunction of all the conditions of the pairs.

4.4 Estimation of gas costs for real smart contracts

The present section illustrates the results given by the gas estimator when used in different contexts, including functions that contain structured for loops and contracts that call other contracts. The results are explored and the greater result is compared to the single estimate given by Remix.

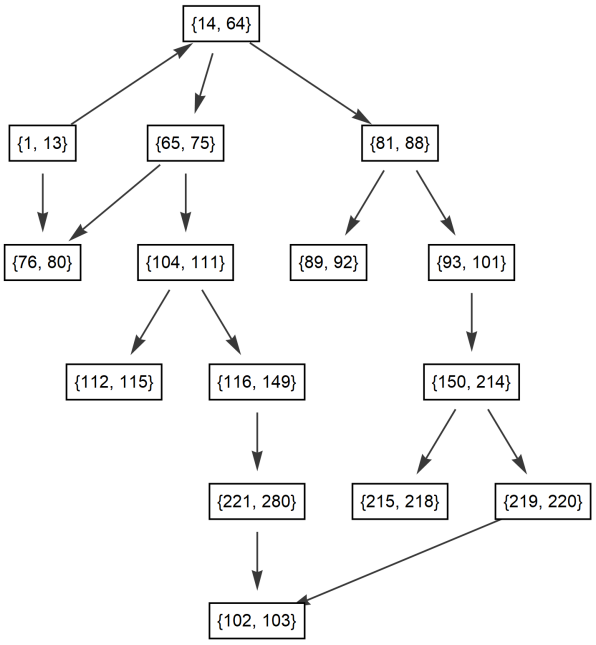
Call to a different contract

This example concerns a situation where an external contract is called. The contract `SendEther` (Figure 4.3a) contains a global variable, `a`, of type `address`, and two functions. `setAddress` receives an address `x` and stores it in variable `a`. `sendEther` calls the contract `a` with the following parameters: 1 ether to be transferred, 10000 units of gas to be spent on the execution of `a`, and empty input. If the transaction is not successful, the method should revert all changes (`revert()`), including the transference of ether. The control flow graph is represented in Figure 4.3b. Blocks are represented by a pair $\{x, y\}$ where x and y are the positions of the instructions where the block starts and ends, respectively.

The analysis of the gas cost of the function `sendEther` returns a result that depends on different types of conditions. In the following, a successful call with parameters $gas, to, va, io, is, oo, os$ is denoted by $\texttt{CALL}(gas, to, va, io, is, oo, os) = 1$, and an unsuccessful call by $\texttt{CALL}(gas, to, va, io, is, oo, os) = 0$. The existence of the account with address a is again represented by $\sigma(a) \neq \bot$, and the inexistence by $\sigma(a) = \bot$. Values are internally represented in the smallest subunit of ether, wei. Since 1 ether is equal to 10^{18} wei, the specified value for the call to the contract a is 10^{18}. This value should not be confused with $\iota.\texttt{value}$, the value of the transaction that directly started the execution of the function `sendEther` of the contract `SendEther`.

- 109 gas if $\iota.\texttt{value} \neq 0$;
- 32618 gas if $\iota.\texttt{value} = 0 \ \wedge \ \texttt{CALL}(10000, a, 10^{18}, 128, 0, 128, 0) = 1 \ \wedge \ \sigma(\texttt{a}) = \bot$;
- 32614 gas if $\iota.\texttt{value} = 0 \ \wedge \ \texttt{CALL}(10000, a, 10^{18}, 128, 0, 128, 0) = 0 \ \wedge \ \sigma(\texttt{a}) = \bot$
- $7618 + \min\Big(10000, gas_i - 10078 - \Big\lfloor \frac{g_i - 10078}{64} \Big\rfloor\Big)$ gas if $\iota.\texttt{value} = 0 \wedge \texttt{CALL}(10000, \texttt{a}, 10^{18}, 128, 0, 128, 0) = 1 \ \wedge \ \sigma(a) \neq \bot$, where g_i is the initial gas given for the execution of `SendEther`;

```
contract SendEther {
    address a;
    function setAddress(address x) {
        a = x;
    }
    function sendEther() {
        if(!a.call.value(1 ether).gas(10000)
             revert();
    }
}
```



(a) `SendEther` contract code

(b) Control flow graph of the contract `SendEther`

Figure 4.3: Source code of the contract `SendEther` and its control flow graph

- $7614 + \min\Big(10000, gas_i - 10078 - \Big\lfloor \frac{g_i - 10078}{64} \Big\rfloor\Big)$ gas if $\iota.\texttt{value} = 0 \wedge \texttt{CALL}(10000, \texttt{a}, 10^{18}, 128, 0, 128, 0) = 0 \wedge \sigma(a) \neq \bot$.

Notice how the cost is much higher if the account a does not exist, since in that case it is necessary to create it. That cost is fixed since the variable part of the cost concerns the gas that is sent to the inner execution, but if the account does not exist that gas is entirely returned to the caller in the end of the (empty) execution. If it does exist, the maximum possible cost is 17618, since no more than the specified 10000 gas can be sent to the inner execution. The slight difference of 4 gas between the estimates for success and insuccess of the call is due to the additional operations needed to revert the operation if that is the case. In case of success, probably not all gas sent is going to be spent, so it is likely that the real cost is lower in that case. If the gas of the call was not specified by the function, then it would not be possible to give a meaningful estimate of the cost because all gas of the execution of `SendEther` could be sent to the inner execution and lost. That is a limitation imposed by the design of the virtual machine and not by the gas estimator. It is possible to obtain a better estimate if one has the code of the contract to call, since in that case it is possible to determine the amount of gas that is returned to the execution of `sendEther`.

Voting smart contract

The following example illustrates one of the previously mentioned applications of Ethereum: an electronic, automatic election. The contract `Voting`, shown in Figure 4.4, is an adaptation of an example contract

present in the Solidity documentation[3]. It contains three global variables, `chairperson`, `voters`, and `proposals`; and three functions, `vote`, `winningProposal`, and `winnerName`, besides the constructor. The keyword `struct` defines a new type of variable. An instance of `Voter` contains a boolean (whether he already voted) and an integer (his vote); an instance of `Proposal` contains a bytearray of length 32 (the name of the candidate) and an integer (his vote count). `Voter`s and `Proposal`s are linked through the mapping `voters`.

The constructor receives an array with the names of three candidates and initialises the variable `proposals` with the specified names and attributing zero votes to each candidate. It also sets `chairperson` to the contract that triggered the execution of the constructor. The function `vote` receives the number of a proposal as input, gets the `Voter` of the address that called it, checks that the voter did not vote already and, if so, sets the vote of the caller to that proposal. The function `winningProposal` returns the number of the proposal with the largest number of votes at the moment, and the function `winnerName` returns the name of that proposal. The contract could implement more features, such as setting a time for the election to end and only allowing the chairperson to change it, giving the possibility to a voter of delegating its vote to other voter (as it is the case in the original contract), attributing different numbers of votes to different voters, or declaring a more complex rule to break ties (in this case it is just the proposal with the smallest index).

As usual, this code was compiled on Remix and all analyses used the resulting bytecode. The control flow graph construction detected the presence of one loop with 3 iterations and a body with two possible paths per iteration, which corresponds to the function `winningProposal`. It did not detect the loop on the constructor function since it was assumed that the contract has already been deployed. The CFG is pictured in Figure 4.5 with the loop guard and body highlighted in red.

We will start by focusing on the function `vote`. The following is a simplification of the output of the gas calculator when applied to this function. The original output does not contain the names of the variables since that information is not a part of the bytecode; it contains instead references to the places where their values are stored by the EVM: for instance, `proposal` is represented as `CALLDATALOAD[4]`, since it starts on the fourth byte of ι.`input`. These references were subsequently substituted by the names of the variables, determined from the documentation of the compiler, for clarity. It is also worth mentioning that some returned conditions were impossible considering information about the types of the variables (information that is also not present in bytecode), so we present here the simplified conditions. Recall from Chapter 2 that ι.`value` and ι.`sender` are, respectively, the address of the contract and the value of the transaction that triggered the execution. All values of variables are relative to their state prior to the current execution of the function.

- 109 gas if ι.`value` $\neq 0$ – the function does not accept transactions with a value different from 0;
- 528 gas if ι.`value` $= 0 \wedge$ `voters[`ι.`sender].voted` – if the caller already voted, the function reverts in the second line, so the expenses are small;
- 30952 gas if ι.`value` $= 0 \wedge$ `!voters[`ι.`sender].voted` $\wedge$ `p` $= 0 \wedge$ `props[p].count` $\neq 0$;

[3]The original contract can be found at `https://solidity.readthedocs.io/en/latest/solidity-by-example.html#voting`

```
contract Voting {
    struct Voter {
        bool voted;
        uint vote;
    }
    struct Proposal {
        bytes32 name;
        uint count;
    }
    address public chairperson;
    mapping(address => Voter) public voters;
    Proposal[3] public props;
    constructor(bytes32[3] proposalNames) public {
        chairperson = msg.sender;
        for (uint i = 0; i < 3; i++) {
            props[i] = Proposal({name: proposalNames[i], count: 0});
        }
    }
    function vote(uint p) public {
        Voter sender = voters[msg.sender];
        require(!sender.voted, "Already voted.");
        sender.voted = true;
        sender.vote = p;
        props[p].count += 1;
    }
    function winningProposal() public view returns (uint winningProposal_) {
        uint winningVoteCount = 0;
        for (uint i = 0; i < 3; i++) {
            if (props[i].count > winningVoteCount) {
                winningVoteCount = props[i].count;
                winningProposal_ = i;
            }
        }
    }
    function winnerName() public view returns (bytes32 winnerName_) {
        winnerName_ = props[winningProposal()].name;
    }
}
```

Figure 4.4: Voting contract code

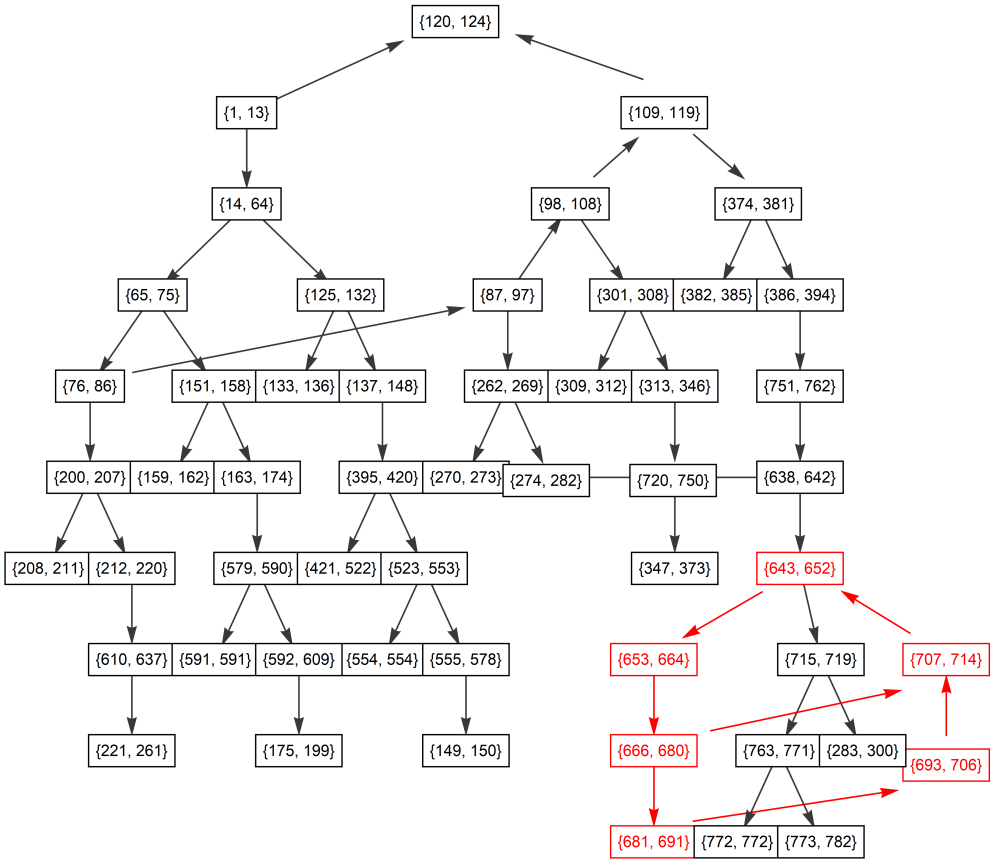


Figure 4.5: Control flow graph of the contract Voting

- 40694 gas if $\iota.\texttt{value} = 0 \wedge \texttt{!voters[}\iota.\texttt{sender]}\texttt{.voted} \wedge \texttt{p} \geq 3$ – this case ends unsuccessfully since the proposal index is greater than the number of proposals, but the gas expenditure is high because that is only detected in the last line of the source code;
- 45952 gas if

$$\iota.\texttt{value} = 0 \wedge \texttt{!voters[}\iota.\texttt{sender]}\texttt{.voted} \wedge \texttt{p} < 3 \wedge (\texttt{props[p]}\texttt{.count} = 0 \wedge (\texttt{p} = 0 \vee \texttt{voters[}\iota.\texttt{sender)]}\texttt{.vote} = 0)) \vee (\texttt{props[p]}\texttt{.count} \neq 0 \wedge \texttt{p} \neq 0 \wedge \texttt{voters[}\iota.\texttt{sender)]}\texttt{.vote} \neq 0))$$

- 60952 gas if $\iota.\texttt{value} = 0 \wedge \texttt{!voters[}\iota.\texttt{sender]}\texttt{.voted} \wedge \texttt{p} \neq 0 \wedge \texttt{p} < 3 \wedge \texttt{props[p]}\texttt{.count} = 0$.

Notice that the successful execution of the third case is less costly than the unsuccessful execution of the fourth case, although the former constitutes an actual vote and the latter has no practical effect. The differences in the costs of the successful cases are due to the opcode `SSTORE`, run to save the votes in the storage of the contract `Voting`. This analysis reveals that the contract is not very fair, since voting in a proposal that has zero votes costs more gas than voting in a proposal that was already voted. This is because substituting a zero value on storage by a non-zero value (in this case, `count`) costs 15000 more gas than substituting a non-zero value by a non-zero value. Remix gives an estimate of 60952 gas for this function.

`winningProposal` is also an interesting function since it contains a for loop with a fixed number of iterations such that a condition is tested in each iteration. The variable `count` of any Proposal should be considered modulo 2^{256}; we omitted that reference for better readability. The other variables are necessarily smaller than 2^{256} because they are given as input either to the function or to the transaction. The output of the gas calculator for this function retrieved, after renaming the variables:

- 175 gas if $\iota.\texttt{value} \neq 0$;
- 1223 gas if $\iota.\texttt{value} = 0 \wedge \texttt{props[0]}\texttt{.count} \leq 0 \wedge \texttt{props[1]}\texttt{.count} \leq 0 \wedge \texttt{props[2]}\texttt{.count} \leq 0$ – the condition tested in the `if` statement is always false, so `winningVoteCount` is always 0. In this case no candidate has votes, so there is no winner. The condition returned has a $\leq$ sign instead of a $=$ sign because the symbolic machine does not know that these values are positive;
- 1482 gas if

  $\iota.\texttt{value} = 0 \wedge$

  $(\texttt{props[0]}\texttt{.count} \leq 0 \wedge \texttt{props[1]}\texttt{.count} \leq 0 \wedge \texttt{props[2]}\texttt{.count} > 0) \vee$

  $(\texttt{props[0]}\texttt{.count} \leq 0 \wedge \texttt{props[1]}\texttt{.count} > 0 \wedge \texttt{props[2]}\texttt{.count} \leq \texttt{props[1]}\texttt{.count}) \vee$

  $(\texttt{props[0]}\texttt{.count} > 0 \wedge \texttt{props[1]}\texttt{.count} \leq \texttt{props[0]}\texttt{.count} \wedge \texttt{props[2]}\texttt{.count} \leq \texttt{props[0]}\texttt{.count}))$

  One of the proposals was once declared as the current winner and did not leave that position until the end of the execution;

- 1741 gas if

  $\iota.\texttt{value} = 0 \wedge$

  $((\texttt{props[0].count} \leq 0 \wedge \texttt{props[2].count} > \texttt{props[1].count} > 0) \vee$

  $(\texttt{props[0].count} > 0 \wedge \texttt{props[1].count} \leq \texttt{props[0].count} \wedge \texttt{props[2].count} > \texttt{props[1].count}) \vee$

  $(\texttt{props[1].count} > \texttt{props[0].count} > 0 \wedge \texttt{props[2].count} \leq \texttt{props[1].count})$

  One of the proposals was once declared as the current winner and was later surpassed by another proposal;

- 2000 gas if $\iota.\texttt{value} = 0 \wedge \texttt{props[2].count} > \texttt{props[1].count} > \texttt{props[0].count} > 0$ – every candidate was the current winner once but the final winner was determined to be the last one.

Notice that the difference between the costs of every consecutive possibilities (except the first pair) is constant. The additional work performed corresponds to updating the value of the variables `winningVoteCount` and `winningProposal` whenever a proposal with a greater number of votes is found. The Remix gas estimate for this function is "infinite" since it contains a loop.

Chapter 5

Conclusions

5.1 Achievements

In the present work we have studied the Ethereum framework, focusing on the Ethereum Virtual Machine, and developed two tools designed to analyse smart contracts.

The Ethereum environment was defined in Chapter 2, where the Ethereum Virtual Machine and the semantics of its operations were explored. In Chapter 2 we have introduced the symbolic EVM, an implementation of the EVM in Mathematica enriched with symbolic features. This machine is able to execute a smart contract with symbolic input and return a complete list of its possible execution paths along with the modifications that were done to the system. We have formally defined the structure of the conditions over input and environmental parameters that characterize each execution path, as well as the features and limitations of the tool. The functionalities of the symbolic machine were illustrated through the analysis of a real smart contract that was exploited in 2018. We were able to retrieve a list of execution paths containing the one that caused the vulnerability.

We have also built a gas estimator for functions of smart contracts. In Chapter 4 we have started by defining the problem and identifying a major difficulty in bytecode analysis: the existence of loops. Then, we have focused on control flow analysis and loop identification and restricted our analysis to programs whose control flow graph is reducible. We have studied, proved the correction, and proved results on the complexity of an algorithm to determine if a control flow graph is reducible, which was included in the broader program that we developed to estimate gas costs. This considers all execution paths and returns an upper estimate on the cost for each path. Our tool is a significant improvement over Remix for three main reasons: it returns a possible cost for each situation, instead of a single upper bound; it considers loops, although it only accepts those that run for a fixed number of iterations; it is able to rule out mutually exclusive conditions that could artificially increase the worst-case gas estimate.

5.2 Future work

Future research directions on this topic could include improvements on the gas estimator with the goal of widening the class of programs that is accepted. Specific suggestions include: considering loops with more than one initialisation block; allowing loops to contain exits mid-loop (typical `break` pattern in most programming languages); including the possibility of specifying array or string sizes as input to the tool, when that is known, so that programs whose loops have a variable number of iterations can be examined in some specific contexts that are relevant to the user.

Another interesting and useful project would be to develop an optimizer of Ethereum bytecode. The Remix compiler generates bytecode that often contains redundancies and poor code ordering, which results in higher gas costs. Code optimization is known to be a hard problem, but there are some strategies that can be applied and proved to maintain the semantics of the original code. In fact, during the present work we have done some experiments on bytecode optimization through the application of a dozen simple rules, such as replacing a sequence of instructions by another cheaper, equivalent sequence of instructions, to bytecode compiled on Remix. We have obtained slightly less expensive code, but it is clear that there is much room for improvement since the optimizer of Remix gives much better results. A more efficient optimization would involve reordering of blocks in order to minimize the number of jumps. The gas estimator would be of great importance in the development of such a tool since it would provide a reliable measure of cost and, consequently, of the performance of the optimizer.

Bibliography

[1] Ackermann, W. (1928). Zum hilbertschen aufbau der reellen zahlen. *Mathematische Annalen*, 99:118–133.

[2] Buterin, V. (2014). A next-generation smart contract and decentralized application platform. Available at: `http://blockchainlab.com/pdf/Ethereum_white_paper-a_next_generation_smart_contract_and_decentralized_application_platform-vitalik-buterin.pdf`.

[3] Chen, T., Li, X., Luo, X., and Zhang, X. (2017). Under-optimized smart contracts devour your money. In *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*, pages 442–446. IEEE.

[4] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2002). *Introduction to Algorithms*, chapter 21. MIT Press.

[5] Dwork, C. and Naor, M. (1992). Pricing via processing or combatting junk mail. In *12th Annual International Cryptology Conference*, pages 139 – 147.

[6] Grishchenko, I., Maffei, M., and Schneidewind, C. (2018). A semantic framework for the security analysis of ethereum smart contracts. In *International Conference on Principles of Security and Trust*, pages 243–269. Springer.

[7] Gunter, C. A. (1992). *Semantics of programming languages: structures and techniques*. MIT Press.

[8] Hecht, M. S. and Ullman, J. D. (1972). Flow graph reducibility. *SIAM J. Comput.*, 1:188–202.

[9] Hecht, M. S. and Ullman, J. D. (1974). Characterizations of reducible flow graphs. *J. ACM*, 21:367–375.

[10] Hildenbrandt, E., Saxena, M., Zhu, X., Rodrigues, N., Daian, P., Guth, D., and Rosu, G. (2017). Kevm: A complete semantics of the ethereum virtual machine. Technical report, University of Illinois.

[11] Hirai, Y. (2017). Defining the ethereum virtual machine for interactive theorem provers. *1st Workshop on Trusted Smart Contracts*.

[12] Luu, L., Chu, D.-H., Olickel, H., Saxena, P., and Hobor, A. (2016). Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, ACM*, pages 254 – 259.

[13] Nakamoto, S. (2009). Bitcoin: A peer-to-peer electronic cash system. Available at: `https://bitcoin.org/bitcoin.pdf`.

[14] Nielson, H. R. and Nielson, F. (1992). *Semantics with applications: a formal introduction.* John Wiley & Sons.

[15] Sernadas, A. and Sernadas, C. (2008). *Foundations of logic and theory of computation.* College Publications.

[16] Szabo, N. (1994). Smart contracts: Building blocks for digital markets. Available at: `http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html`.

[17] Tarjan, R. E. (1974). Testing flow graph reducibility. *Journal of Computer and System Sciences*, 9:355–365.

[18] Tarjan, R. E. (1985). Amortized computational complexity. *SIAM. J. on Algebraic and Discrete Methods*, 6:306–318.

[19] Wood, G. (2014). Ethereum: A secure decentralized generalized transaction ledger. Available at: `https://gavwood.com/paper.pdf`.

Appendix A

Semantics of the symbolic EVM

This appendix contains a comprehensive list of the semantics of the operations of the symbolic EVM, greatly inspired by the original semantics [6] and already sketched in Chapter 3. We start by defining some auxiliary functions related to gas costs.

A.1 Auxiliary functions

$M\colon (\mathbb{N}_{256})^3 \to \mathbb{N}_{256}$ is a function that helps to update the number of active words in memory. It receives the current number of active words in memory, i, the offset o and the size s of the portion of the memory that is being considered (either to read or to write). It there is no memory accessed (size equal to 0), it outputs the original number of active words in memory. Otherwise, it returns the word corresponding to last accessed position of the memory (considering that a word is a 32-bit value).

$$M(i,o,s) = \begin{cases} i & \text{if } s = 0 \\ \max\left(i, \left\lceil \frac{o+s}{32} \right\rceil\right) & \text{otherwise} \end{cases} \tag{A.1}$$

$C_{mem}\colon (\mathbb{N}_{256})^2 \to \mathbb{Z}$ is an auxiliary function used to compute gas costs due to memory usage, comparing the new and the old values of active words in memory.

$$C_{mem}(i,i') = 3(i'-i) + \left\lfloor \frac{i'^2}{512} \right\rfloor - \left\lfloor \frac{i^2}{512} \right\rfloor \tag{A.2}$$

$C_{base}\colon \mathbb{N}_{256} \times \{0,1\} \to \mathbb{N}_{256}$ and $C_{extra}\colon \mathbb{N}_{256} \times \{0,1\} \to \mathbb{N}_{256}$ help to calculate the cost of calling a contract and concern the contribution of the value and of the existence of the called account.

$$C_{base}(va, flag) = \begin{cases} 25700 & \text{if } va = 0 \text{ and } flag = 0 \\ 700 & \text{if } va = 0 \text{ and } flag \neq 0 \\ 32200 & \text{if } va \neq 0 \text{ and } flag = 0 \\ 7200 & \text{if } va \neq 0 \text{ and } flag \neq 0 \end{cases} \tag{A.3}$$

$$C_{extra}(va, flag) = \begin{cases} 25700 & \text{if } va = 0 \text{ and } flag = 0 \\ 700 & \text{if } va = 0 \text{ and } flag \neq 0 \\ 34700 & \text{if } va \neq 0 \text{ and } flag = 0 \\ 9700 & \text{if } va \neq 0 \text{ and } flag \neq 0 \end{cases} \tag{A.4}$$

$C_{gascap}\colon \mathbb{N}_{256} \times \{0,1\} \times \mathbb{N}_{256} \times \mathbb{N}_{256} \to \mathbb{N}_{256}$ is also used when calling other contract and considers both the gas set by the user and the available gas for the execution.

$$C_{gascap}(va, flag, g, gas) = \begin{cases} g & \text{if } C_{extra}(va, flag) > gas \text{ and } va = 0 \\ g + 2300 & \text{if } C_{extra}(va, flag) > gas \text{ and } va \neq 0 \\ \min(g, L(gas - C_{extra}(va, flag))) & \text{if } C_{extra}(va, flag) \leq gas \text{ and } va = 0 \\ \min(g, L(gas - C_{extra}(va, flag))) + 2300 & \text{if } C_{extra}(va, flag) \leq gas \text{ and } va \neq 0 \end{cases} \tag{A.5}$$

Finally, $L\colon \mathbb{N}_{256} \to \mathbb{N}_{256}$ is an auxiliary function used to compute C_{extra}.

$$L(n) = n - \left\lfloor \frac{n}{64} \right\rfloor \mod 2^{256} \tag{A.6}$$

A.2 Stop and arithmetic operations

The `STOP` instruction only needs a small change in order to accommodate the modification in $HALT$.

$$\frac{\omega_{\mu,\iota} = \texttt{STOP} \quad g = \mu.\texttt{gas} \quad \mathbf{cond} = \mu.\texttt{cond}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to HALT(\sigma, g, \epsilon, \eta, \mathbf{cond}) :: S}$$

The modifications on simple arithmetic operations are trivial. If $i_{arith} \in \{\texttt{ADD}, \texttt{MUL}, \texttt{SUB}\}$ and

$$\text{fun}_{arith}(i_{arith}) = \begin{cases} \lambda.(\mathbf{a}, \mathbf{b}).\mathbf{a} + \mathbf{b} \mod 2^{256}, & i_{arith} = \texttt{ADD} \\ \lambda.(\mathbf{a}, \mathbf{b}).\mathbf{a} - \mathbf{b} \mod 2^{256}, & i_{arith} = \texttt{MUL} \\ \lambda.(\mathbf{a}, \mathbf{b}).\mathbf{a} \times \mathbf{b} \mod 2^{256}, & i_{arith} = \texttt{SUB} \end{cases}$$

we can write a general binary arithmetic operation rule

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = i_{arith} \qquad valid(\mu.\texttt{gas}, 3, |\mathbf{s}|) \qquad \mu.\mathbf{s} = \mathbf{a} :: \mathbf{b} :: \mathbf{s} \\ \mu' = \mu[\texttt{gas} - = 3][\texttt{pc} + = 1][\mathbf{s} \to \text{fun}_{arith}(i_{arith})(\mathbf{a}, \mathbf{b}) :: \mathbf{s}]\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

If the stack does not have enough elements or there is not enough gas, an exception is thrown. Notice that the exception returns the current condition.

$$\frac{\omega_{\mu,\iota} = i_{arith} \quad (\neg valid(\mu.\texttt{gas}, 3, |\mu.\mathbf{s}|) \vee |\mu.\mathbf{s}| < 2) \quad \mathbf{cond} = \mu.\texttt{cond}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to EXC(\mathbf{cond}) :: S}$$

The operations DIV and MOD have a different behaviour depending on whether the second argument is zero or not. If $i_{div} \in \{\texttt{DIV}, \texttt{MOD}\}$ and

$$\text{fun}_{div}(i_{div}) = \begin{cases} \lambda.(\mathbf{a}, \mathbf{b}).\lfloor \mathbf{a} \div \mathbf{b} \rfloor, & i_{div} = \texttt{DIV} \\ \lambda.(\mathbf{a}, \mathbf{b}).\mathbf{a} \mod \mathbf{b}, & i_{div} = \texttt{MOD} \end{cases}$$

then the general rule can be written as:

$$\frac{\begin{array}{c} \omega_{\mu,\iota} = i_{div} \qquad valid(\mu.\texttt{gas}, 5, |\mathbf{s}|) \qquad \mu.\mathbf{s} = \mathbf{a} :: \mathbf{b} :: \mathbf{s} \\ \mu_1' = \mu[\texttt{gas}- = 5][\texttt{pc}+ = 1][\mathbf{s} \to 0 :: \mathbf{s}][\texttt{cond} \to \mu.\texttt{cond} \wedge \mathbf{b} = 0] \\ \mu_2' = \mu[\texttt{gas}- = 5][\texttt{pc}+ = 1][\mathbf{s} \to \text{fun}_{div}(i_{div})(\mathbf{a}, \mathbf{b}) :: \mathbf{s}][\texttt{cond} \to \mu.\texttt{cond} \wedge \mathbf{b} \neq 0] \end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to \begin{array}{c} (\mu_1', \iota, \sigma, \eta) :: S \\ (\mu_2', \iota, \sigma, \eta) :: S \end{array}}$$

$$\frac{\omega_{\mu,\iota} = i_{div} \qquad (\neg valid(\mu.\texttt{gas}, 5, |\mu.\mathbf{s}|) \vee |\mu.\mathbf{s}| < 2) \qquad \mathbf{cond} = \mu.\texttt{cond}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to EXC(\mathbf{cond}) :: S}$$

ADDMOD and MULMOD have similar semantics but take three arguments. If $i_{mod} \in \{\texttt{ADDMOD}, \texttt{MULMOD}\}$ and

$$\text{fun}_{mod}(i_{mod}) = \begin{cases} \lambda.(\mathbf{a}, \mathbf{b}, \mathbf{c}).\mathbf{a} + \mathbf{b} \mod \mathbf{c}, & i_{mod} = \texttt{ADDMOD} \\ \lambda.(\mathbf{a}, \mathbf{b}, \mathbf{c}).\mathbf{a} \times \mathbf{b} \mod \mathbf{c}, & i_{mod} = \texttt{MULMOD} \end{cases}$$

then the general rule can be written as:

$$\frac{\begin{array}{c} \omega_{\mu,\iota} = i_{mod} \qquad valid(\mu.\texttt{gas}, 8, |\mathbf{s}|) \qquad \mu.\mathbf{s} = \mathbf{a} :: \mathbf{b} :: \mathbf{c} :: \mathbf{s} \\ \mu_1' = \mu[\texttt{gas}- = 8][\texttt{pc}+ = 1][\mathbf{s} \to 0 :: \mathbf{s}][\texttt{cond} \to \mu.\texttt{cond} \wedge \mathbf{c} = 0] \\ \mu_2' = \mu[\texttt{gas}- = 8][\texttt{pc}+ = 1][\mathbf{s} \to \text{fun}_{mod}(i_{mod})(\mathbf{a}, \mathbf{b}, \mathbf{c}) :: \mathbf{s}][\texttt{cond} \to \mu.\texttt{cond} \wedge \mathbf{c} \neq 0] \end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to \begin{array}{c} (\mu_1', \iota, \sigma, \eta) :: S \\ (\mu_2', \iota, \sigma, \eta) :: S \end{array}}$$

$$\frac{\omega_{\mu,\iota} = i_{mod} \qquad (\neg valid(\mu.\texttt{gas}, 8, |\mu.\mathbf{s}|) \vee |\mu.\mathbf{s}| < 3) \qquad \mathbf{cond} = \mu.\texttt{cond}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to EXC(\mathbf{cond}) :: S}$$

The exponential operation EXP is an exceptional case of an arithmetic operation where we do not allow all the arguments to be symbolic. The base can be a symbol, but the exponent must be a number. The gas cost of this instruction depends logarithmically on the exponent and, so, there is an infinite number of possibilities for the result. Since the gas must be numeric, the exponent must be numeric too.[1]

$$\frac{\begin{array}{c} \omega_{\mu,\iota} = \texttt{EXP} \qquad valid(\mu.\texttt{gas}, c, |\mathbf{s}| + 1) \\ \mu.\mathbf{s} = \mathbf{a} :: b :: \mathbf{s} \qquad c = (b = 0) \ ? \ 10 : 10 + 50 \times (1 + \lfloor \log_{256} b \rfloor) \\ \mathbf{x} = \mathbf{a}^b \mod 2^{256} \qquad \mu' = \mu[\texttt{gas}- = c][\texttt{pc}+ = 1][\mathbf{s} \to \mathbf{x} :: \mathbf{s}] \end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

[1]The cost of this operation does not match the yellow paper [19] since it was later changed as a part of the Spurious Dragon Hard Fork: https://blog.ethereum.org/2016/11/18/hard-fork-no-4-spurious-dragon/

$$\frac{\omega_{\mu,\iota} = \texttt{EXP} \qquad c = (b = 0)\ ?\ 10 : 10 + 50 \times (1 + \lfloor \log_{256} b \rfloor) \\ \mu.\mathbf{s} = \mathbf{a} :: b :: \mathbf{s} \qquad \neg valid(\mu.\texttt{gas}, c, |\mathbf{s}| + 1) \qquad \mathbf{cond} = \mu.\texttt{cond}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to EXC(\mathbf{cond}) :: S}$$

$$\frac{\omega_{\mu,\iota} = \texttt{EXP} \qquad |\mu.\mathbf{s}| < 2}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to EXC(\mu.\texttt{cond}) :: S}$$

A.3 Bitwise and signed integer operations

The operations SDIV, SMOD, SIGNEXTEND, and BYTE deal with the binary digits of their arguments. When the arguments are symbolic we chose to leave a symbol representing the unevaluated functions. If $i_{sign} = \{\texttt{SDIV}, \texttt{SMOD}\}$ and

$$\text{fun}_{sign}(i_{sign}) = \begin{cases} \lambda.(\mathbf{a}, \mathbf{b}).\mathbf{sdiv(a,b)}, & i_{sign} = \texttt{SDIV} \\ \lambda.(\mathbf{a}, \mathbf{b}).\mathbf{smod(a,b)}, & i_{sign} = \texttt{SMOD} \end{cases}$$

then the general rule can be written as:

$$\frac{\omega_{\mu,\iota} = i_{sign} \qquad valid(\mu.\texttt{gas}, 5, |\mathbf{s}| + 1) \qquad \mu.\mathbf{s} = \mathbf{a} :: 0 :: \mathbf{s} \\ \mu' = \mu[\texttt{gas}- = 5][\texttt{pc}+ = 1][\texttt{s} \to 0 :: \mathbf{s}]}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

SDIV contains a special rule for the case where the first argument is the two's complement representation of -2^{255}, 2^{255}, and the second argument is the two's complement representation of -1, $(-1)^-$.

$$\frac{\omega_{\mu,\iota} = \texttt{SDIV} \qquad valid(\mu.\texttt{gas}, 5, |\mathbf{s}| + 1) \qquad \mu.\mathbf{s} = 2^{255} :: (-1)^- :: \mathbf{s} \\ \mu' = \mu[\texttt{gas}- = 5][\texttt{pc}+ = 1][\texttt{s} \to 2^{255} :: \mathbf{s}]}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = i_{sign} \qquad valid(\mu.\texttt{gas}, 5, |\mathbf{s}| + 1) \qquad \mu.\mathbf{s} = \mathbf{a} :: \mathbf{b} :: \mathbf{s} \\ \mu' = \mu[\texttt{gas}- = 5][\texttt{pc}+ = 1][\texttt{s} \to \text{fun}_{sign}(i_{sign})(\mathbf{a}, \mathbf{b}) :: \mathbf{s}]}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = i_{sign} \qquad (\neg valid(\mu.\texttt{gas}, 5, |\mu.\mathbf{s}|) \vee |\mu.\mathbf{s}| < 2) \qquad \mathbf{cond} = \mu.\texttt{cond}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to EXC(\mathbf{cond}) :: S}$$

$$\frac{\omega_{\mu,\iota} = \texttt{BYTE} \qquad valid(\mu.\texttt{gas}, 3, |\mathbf{s}| + 1) \\ \mu.\mathbf{s} = a :: b :: \mathbf{s} \qquad x = b[8a, 8a + 7] \cdot 0^{248} \\ \mu' = \mu[\texttt{gas}- = 3][\texttt{pc}+ = 1][\texttt{s} \to x :: \mathbf{s}]}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \texttt{BYTE} \quad valid(\mu.\texttt{gas}, 3, |\mathbf{s}| + 1) \quad \mu.\mathbf{s} = \mathbf{a} :: \mathbf{b} :: \mathbf{s} \quad \mu.\texttt{cond} \Rightarrow \mathbf{b} \geq 32 \quad \mu' = \mu[\texttt{gas} - = 3][\texttt{pc} + = 1][\texttt{s} \rightarrow 0 :: \mathbf{s}]}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \rightarrow (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \texttt{BYTE} \quad valid(\mu.\texttt{gas}, 3, |\mathbf{s}| + 1) \quad \mu.\mathbf{s} = \mathbf{a} :: \mathbf{b} :: \mathbf{s} \quad \mu' = \mu[\texttt{gas} - = 3][\texttt{pc} + = 1][\texttt{s} \rightarrow \mathbf{byte(a,b)} :: \mathbf{s}]}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \rightarrow (\mu', \iota, \sigma, \eta) :: S}$$

A.4 Comparison operations

Comparison operations like `LT`, `GT`, `SLT`, `SGT`, and `EQ` take two elements and write 1 or 0 in the execution stack depending on the result of the comparison. If $i_{comp} \in \{\texttt{LT}, \texttt{GT}, \texttt{SLT}, \texttt{SGT}, \texttt{EQ}\}$, then a general rule can be written as:

$$\frac{\omega_{\mu,\iota} = i_{comp} \quad valid(\mu.\texttt{gas}, 3, |\mathbf{s}| + 1) \quad \mu.\mathbf{s} = \mathbf{a} :: \mathbf{b} :: \mathbf{s} \quad \mu_1' = \mu[\texttt{gas} - = 3][\texttt{pc} + = 1][\texttt{s} \rightarrow 0 :: \mathbf{s}][\texttt{cond} \rightarrow \mu.\texttt{cond} \wedge cond_0] \quad \mu_2' = \mu[\texttt{gas} - = 3][\texttt{pc} + = 1][\texttt{s} \rightarrow 1 :: \mathbf{s}][\texttt{cond} \rightarrow \mu.\texttt{cond} \wedge cond_1]}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \rightarrow \begin{array}{l}(\mu_1', \iota, \sigma, \eta) :: S \\ (\mu_2', \iota, \sigma, \eta) :: S\end{array}}$$

where $cond_0$ and $cond_1$ are formulas that depend on the opcode:

$$cond_0 = \begin{cases} \mathbf{a} \geq \mathbf{b} & \text{if } i_{comp} = \texttt{LT} \\ \mathbf{a} \leq \mathbf{b} & \text{if } i_{comp} = \texttt{GT} \\ TC(\mathbf{a}) \geq TC(\mathbf{b}) & \text{if } i_{comp} = \texttt{SLT} \\ TC(\mathbf{a}) \leq TC(\mathbf{b}) & \text{if } i_{comp} = \texttt{SGT} \\ \mathbf{a} \neq \mathbf{b} & \text{if } i_{comp} = \texttt{EQ} \end{cases} \tag{A.7}$$

$$cond_1 = \begin{cases} \mathbf{a} < \mathbf{b} & \text{if } i_{comp} = \texttt{LT} \\ \mathbf{a} > \mathbf{b} & \text{if } i_{comp} = \texttt{GT} \\ TC(\mathbf{a}) < TC(\mathbf{b}) & \text{if } i_{comp} = \texttt{SLT} \\ TC(\mathbf{a}) > TC(\mathbf{b}) & \text{if } i_{comp} = \texttt{SGT} \\ \mathbf{a} = \mathbf{b} & \text{if } i_{comp} = \texttt{EQ} \end{cases} \tag{A.8}$$

$$\frac{\omega_{\mu,\iota} = i_{comp} \quad (\neg valid(\mu.\texttt{gas}, 3, |\mu.\mathbf{s}| + 1) \vee |\mu.\mathbf{s}| < 2) \quad \mathbf{cond} = \mu.\texttt{cond}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \rightarrow EXC(\mathbf{cond}) :: S}$$

`ISZERO` writes 1 to the stack if its top element is 0 and 0 otherwise. If the top element is a symbol, it creates two copies of the current call stack.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{ISZERO} \qquad valid(\mu.\texttt{gas}, 3, |\mathbf{s}|) \qquad \mu.\mathbf{s} = \mathbf{a} :: \mathbf{s} \\ \mu'_1 = \mu[\texttt{gas} - = 3][\texttt{pc} + = 1][\mathbf{s} \to 1 :: \mathbf{s}][\texttt{cond} \to \mu.\texttt{cond} \wedge \mathbf{a} = 0] \\ \mu'_2 = \mu[\texttt{gas} - = 3][\texttt{pc} + = 1][\mathbf{s} \to 0 :: \mathbf{s}][\texttt{cond} \to \mu.\texttt{cond} \wedge \mathbf{a} \neq 0]\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to \begin{array}{c}(\mu'_1, \iota, \sigma, \eta) :: S \\ (\mu'_2, \iota, \sigma, \eta) :: S\end{array}}$$

$$\frac{\omega_{\mu,\iota} = \texttt{ISZERO} \qquad (\neg valid(\mu.\texttt{gas}, 3, |\mu.\mathbf{s}|) \vee |\mu.\mathbf{s}| < 1) \qquad \mathbf{cond} = \mu.\texttt{cond}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to EXC(\mathbf{cond}) :: S}$$

Bitwise operations cannot be resolved if one of their arguments is symbolic. However, we can use information contained in the function symbols *isAddress* and *isBool*, and the knowledge that all integers can be represented by at most 256 bits, to simplify some cases.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{AND} \qquad \mu.\mathbf{s} = 0 :: \mathbf{b} :: \mathbf{s} \qquad valid(\mu.\texttt{gas}, 3, |\mathbf{s}| + 1) \\ \mu' = \mu[\texttt{gas} \ - = 3]][\texttt{pc} \ + = 1][\mathbf{s} \to 0 :: \mathbf{s}]\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{AND} \qquad \mu.\mathbf{s} = \mathbf{a} :: 0 :: \mathbf{s} \qquad valid(\mu.\texttt{gas}, 3, |\mathbf{s}| + 1) \\ \mu' = \mu[\texttt{gas} \ - = 3]][\texttt{pc} \ + = 1][\mathbf{s} \to 0 :: \mathbf{s}]\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{AND} \qquad \mu.\mathbf{s} = 1 :: \mathbf{b} :: \mathbf{s} \qquad valid(\mu.\texttt{gas}, 3, |\mathbf{s}| + 1) \\ \mu.\texttt{cond} \Rightarrow isBool(\mathbf{b}) \qquad \mu' = \mu[\texttt{gas} \ - = 3]][\texttt{pc} \ + = 1][\mathbf{s} \to \mathbf{b} :: \mathbf{s}]\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{AND} \qquad \mu.\mathbf{s} = \mathbf{a} :: 1 :: \mathbf{s} \qquad valid(\mu.\texttt{gas}, 3, |\mathbf{s}| + 1) \\ \mu.\texttt{cond} \Rightarrow isBool(\mathbf{a}) \qquad \mu' = \mu[\texttt{gas} \ - = 3]][\texttt{pc} \ + = 1][\mathbf{s} \to \mathbf{a} :: \mathbf{s}]\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{AND} \qquad \mu.\mathbf{s} = 2^{160} - 1 :: \mathbf{b} :: \mathbf{s} \qquad valid(\mu.\texttt{gas}, 3, |\mathbf{s}| + 1) \\ \mu.\texttt{cond} \Rightarrow isAddress(\mathbf{b}) \qquad \mu' = \mu[\texttt{gas} \ - = 3]][\texttt{pc} \ + = 1][\mathbf{s} \to \mathbf{b} :: \mathbf{s}]\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{AND} \qquad \mu.\mathbf{s} = \mathbf{a} :: 2^{160} - 1 :: \mathbf{s} \qquad valid(\mu.\texttt{gas}, 3, |\mathbf{s}| + 1) \\ \mu.\texttt{cond} \Rightarrow isAddress(\mathbf{a}) \qquad \mu' = \mu[\texttt{gas} \ - = 3]][\texttt{pc} \ + = 1][\mathbf{s} \to \mathbf{a} :: \mathbf{s}]\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{AND} \qquad \mu.\mathbf{s} = 2^{256} - 1 :: \mathbf{b} :: \mathbf{s} \qquad valid(\mu.\texttt{gas}, 3, |\mathbf{s}| + 1) \\ \mu' = \mu[\texttt{gas} \ - = 3]][\texttt{pc} \ + = 1][\mathbf{s} \to \mathbf{b} :: \mathbf{s}]\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{AND} \qquad \mu.\mathbf{s} = \mathbf{a} :: 2^{256} - 1 :: \mathbf{s} \qquad valid(\mu.\texttt{gas}, 3, |\mathbf{s}| + 1) \\ \mu' = \mu[\texttt{gas} \ - = 3]][\texttt{pc} \ + = 1][\mathbf{s} \to \mathbf{a} :: \mathbf{s}]\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \texttt{AND} \quad \mu.\mathbf{s} = \mathbf{a} :: \mathbf{b} :: \mathbf{s} \quad \mathit{valid}(\mu.\texttt{gas}, 3, |\mathbf{s}| + 1) \quad \mu' = \mu[\texttt{gas} \ -= 3]][\texttt{pc} \ += 1][\mathbf{s} \to \mathbf{a} \& \mathbf{b} :: \mathbf{s}]}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

`OR` is analogous to `AND`.

$$\frac{\omega_{\mu,\iota} = \texttt{OR} \quad \mu.\mathbf{s} = 0 :: \mathbf{b} :: \mathbf{s} \quad \mathit{valid}(\mu.\texttt{gas}, 3, |\mathbf{s}| + 1) \quad \mu' = \mu[\texttt{gas} \ -= 3]][\texttt{pc} \ += 1][\mathbf{s} \to \mathbf{b} :: \mathbf{s}]}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \texttt{OR} \quad \mu.\mathbf{s} = \mathbf{a} :: 0 :: \mathbf{s} \quad \mathit{valid}(\mu.\texttt{gas}, 3, |\mathbf{s}| + 1) \quad \mu' = \mu[\texttt{gas} \ -= 3]][\texttt{pc} \ += 1][\mathbf{s} \to \mathbf{a} :: \mathbf{s}]}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \texttt{OR} \quad \mu.\mathbf{s} = 1 :: \mathbf{b} :: \mathbf{s} \quad \mathit{valid}(\mu.\texttt{gas}, 3, |\mathbf{s}| + 1) \quad \mu.\texttt{cond} \Rightarrow \mathit{isBool}(\mathbf{b}) \quad \mu' = \mu[\texttt{gas} \ -= 3]][\texttt{pc} \ += 1][\mathbf{s} \to 1 :: \mathbf{s}]}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \texttt{OR} \quad \mu.\mathbf{s} = \mathbf{a} :: 1 :: \mathbf{s} \quad \mathit{valid}(\mu.\texttt{gas}, 3, |\mathbf{s}| + 1) \quad \mu.\texttt{cond} \Rightarrow \mathit{isBool}(\mathbf{a}) \quad \mu' = \mu[\texttt{gas} \ -= 3]][\texttt{pc} \ += 1][\mathbf{s} \to 1 :: \mathbf{s}]}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \texttt{OR} \quad \mu.\mathbf{s} = 2^{160} - 1 :: \mathbf{b} :: \mathbf{s} \quad \mathit{valid}(\mu.\texttt{gas}, 3, |\mathbf{s}| + 1) \quad \mu.\texttt{cond} \Rightarrow \mathit{isAddress}(\mathbf{b}) \quad \mu' = \mu[\texttt{gas} \ -= 3]][\texttt{pc} \ += 1][\mathbf{s} \to 2^{160} - 1 :: \mathbf{s}]}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \texttt{OR} \quad \mu.\mathbf{s} = \mathbf{a} :: 2^{160} - 1 :: \mathbf{s} \quad \mathit{valid}(\mu.\texttt{gas}, 3, |\mathbf{s}| + 1) \quad \mu.\texttt{cond} \Rightarrow \mathit{isAddress}(\mathbf{a}) \quad \mu' = \mu[\texttt{gas} \ -= 3]][\texttt{pc} \ += 1][\mathbf{s} \to 2^{160} - 1 :: \mathbf{s}]}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \texttt{OR} \quad \mu.\mathbf{s} = 2^{256} - 1 :: \mathbf{b} :: \mathbf{s} \quad \mathit{valid}(\mu.\texttt{gas}, 3, |\mathbf{s}| + 1) \quad \mu' = \mu[\texttt{gas} \ -= 3]][\texttt{pc} \ += 1][\mathbf{s} \to 2^{256} - 1 :: \mathbf{s}]}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \texttt{OR} \quad \mu.\mathbf{s} = \mathbf{a} :: 2^{256} - 1 :: \mathbf{s} \quad \mathit{valid}(\mu.\texttt{gas}, 3, |\mathbf{s}| + 1) \quad \mu' = \mu[\texttt{gas} \ -= 3]][\texttt{pc} \ += 1][\mathbf{s} \to 2^{256} - 1 :: \mathbf{s}]}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

`XOR` does not allow any simplification.

$$\frac{\omega_{\mu,\iota} = \texttt{XOR} \quad \mu.\mathbf{s} = \mathbf{a} :: \mathbf{b} :: \mathbf{s} \quad \mathit{valid}(\mu.\texttt{gas}, 3, |\mathbf{s}| + 1) \quad \mu' = \mu[\texttt{gas} \ -= 3]][\texttt{pc} \ += 1][\mathbf{s} \to \mathbf{a} \oplus \mathbf{b} :: \mathbf{s}]}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{(\omega_{\mu,\iota} = \texttt{AND} \vee \omega_{\mu,\iota} = \texttt{OR} \vee \omega_{\mu,\iota} = \texttt{XOR}) \quad (\neg valid(\mu.\texttt{gas}, 3, |\mu.\texttt{s}|) \vee |\mu.\texttt{s}| < 2) \quad \textbf{cond} = \mu.\texttt{cond}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to EXC(\textbf{cond}) :: S}$$

`NOT` can be simplified noticing that the universe is $\mathbb{N}_{256}$.

$$\frac{\omega_{\mu,\iota} = \texttt{NOT} \quad \mu.\texttt{s} = \textbf{a} :: \textbf{s} \quad valid(\mu.\texttt{gas}, 3, |\textbf{s}| + 1) \quad \mu' = \mu[\texttt{gas} \ -= 3]][\texttt{pc} \ += 1][\texttt{s} \to 2^{256} - 1 - \textbf{a} :: \textbf{s}]}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \texttt{NOT} \quad (\neg valid(\mu.\texttt{gas}, 3, |\mu.\texttt{s}|) \vee |\mu.\texttt{s}| < 1) \quad \textbf{cond} = \mu.\texttt{cond}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to EXC(\textbf{cond}) :: S}$$

A.5 Environmental information

The operations `CALLDATASIZE`, and `CODESIZE` receive no parameters and return numeric values because it would not make sense for them to return symbols. `EXTCODESIZE` receives a numerical address and retrieves a number. `ADDRESS`, `ORIGIN`, and `CALLER` return an address that may be symbolic, as long as the symbol is stored in the global state σ. The semantic rules of these operations are as expected.

`BALANCE` receives an address that may be symbolic, and may return a symbolic balance.

$$\frac{\omega_{\mu,\iota} = \texttt{BALANCE} \quad \mu.\texttt{s} = \textbf{a} :: \textbf{s} \quad valid(\mu.\texttt{gas}, 400, |\textbf{s}| + 1) \quad \textbf{ba} = (\sigma(\textbf{a} \mod 2^{160}) = (n, \textbf{b}, code, stor)) \ ? \ \textbf{b} \ : \ 0 \quad \mu' = \mu[\texttt{gas} \ -= 400]][\texttt{pc} \ += 1][\texttt{s} \to \textbf{ba} :: \textbf{s}]}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \texttt{BALANCE} \quad (\neg valid(\mu.\texttt{gas}, 400, |\mu.\texttt{s}|) \vee |\mu.\texttt{s}| < 1) \quad \textbf{cond} = \mu.\texttt{cond}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to EXC(\textbf{cond}) :: S}$$

`CALLVALUE` may return a symbol.

$$\frac{\omega_{\mu,\iota} = \texttt{CALLVALUE} \quad valid(\mu.\texttt{gas}, 2, |\textbf{s}|) \quad \mu' = \mu[\texttt{gas} \ -= 2]][\texttt{pc} \ += 1][\texttt{s} \to \iota.\textbf{value} :: \textbf{s}]}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \texttt{CALLVALUE} \quad \neg valid(\mu.\texttt{gas}, 2, |\mu.\texttt{s}| + 1) \quad \textbf{cond} = \mu.\texttt{cond}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to EXC(\textbf{cond}) :: S}$$

`CALLDATALOAD` is a special case. The structure of the ι.`input` field is the following: the first 4 bytes are the last 4 bytes of the hash of the function we wish to call; the next 32 bytes are the first argument to be passed to the function, the next 32 bytes are the second argument to be passed to the function, and so on. Our analysis intends to monitorize the behaviour of every possible call to a contract, that is, every possible call to a function of a contract. So, the hash of the function must be numeric – we need

to know which function we are trying to execute. However, in order to include all execution paths we allow the arguments to be symbolic. A symbolic argument is passed in the same way a symbolic element is stored in and retrieved from the memory: the 32 corresponding bytes have the same symbol.

The motivation for these rules was already detailed in Section 3.2.2. The rules consider that the numbers have 256 bits. Notice that the usual math font states that the values are numeric. The first rule concerns the successful request for the contents of the beginning of $\iota.\texttt{input}$.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{CALLDATALOAD} \\ \mu.\mathbf{s} = 0 :: \mathbf{s} \qquad valid(\mu.\texttt{gas}, 3, |\mathbf{s}| + 1) \qquad k = min(|\iota.\texttt{input}|, 4) \\ d = \iota.\texttt{input}[0, k-1] \qquad d' = d \cdot 0^{224+32-k} \qquad \mu' = \mu[\texttt{gas} - = 3][\texttt{pc} + = 1][\mathbf{s} \to d' :: \mathbf{s}]\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

Recall that $\mathbb{N}_8$ is the set of bytes.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{CALLDATALOAD} \\ \mu.\mathbf{s} = 0 :: \mathbf{s} \qquad valid(\mu.\texttt{gas}, 3, |\mathbf{s}| + 1) \qquad k = min(|\iota.\texttt{input}|, 4) \\ \mathbf{d} = \iota.\texttt{input}[0, k-1] \qquad \mathbf{d} \notin \mathbb{N}_8 \qquad \mathbf{cond} = \mu.\texttt{cond}\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to EXC(\mathbf{cond}) :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{CALLDATALOAD} \\ \mu.\mathbf{s} = a :: \mathbf{s} \qquad a \equiv 4 \mod 32 \qquad valid(\mu.\texttt{gas}, 3, |\mathbf{s}| + 1) \qquad k = min(|\iota.\texttt{input}| - a, 32) \\ d = \iota.\texttt{input}[a, a + k - 1] \qquad d' = d \cdot 0^{256-8 \cdot k} \qquad \mu' = \mu[\texttt{gas} - = 3][\texttt{pc} + = 1][\mathbf{s} \to d' :: \mathbf{s}]\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

The following rule only applies to symbols. Only the case where the input has the desired size is accepted.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{CALLDATALOAD} \\ \mu.\mathbf{s} = a :: \mathbf{s} \qquad a \equiv 4 \mod 32 \qquad valid(\mu.\texttt{gas}, 3, |\mathbf{s}| + 1) \qquad |\iota.\texttt{input}| \geq a + 31 \\ \iota.\texttt{input}[a] = \cdots = \iota.\texttt{input}[a + 31] = \mathbf{d} \qquad \mathbf{d} \notin \mathbb{N}_8 \qquad \mu' = \mu[\texttt{gas} - = 3][\texttt{pc} + = 1][\mathbf{s} \to \mathbf{d} :: \mathbf{s}]\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

At least one of the bytes is a symbol and the bytes are not all equal.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{CALLDATALOAD} \\ \mu.\mathbf{s} = a :: \mathbf{s} \qquad a \equiv 4 \mod 32 \qquad valid(\mu.\texttt{gas}, 3, |\mathbf{s}| + 1) \qquad k = min(|\iota.\texttt{input}| - a, 32) \\ \exists i \in \{0, \ldots, k-1\} : \iota.\texttt{input}[a + i] \notin \mathbb{N}_8 \qquad \neg(\iota.\texttt{input}[a] = \cdots = \iota.\texttt{input}[a + k - 1]) \qquad \mathbf{cond} = \mu.\texttt{cond}\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to EXC(\mathbf{cond}) :: S}$$

The input contains symbols and does not have the desired size, therefore the 32 bytes cannot be equal.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{CALLDATALOAD} \\ \mu.\mathbf{s} = a :: \mathbf{s} \qquad a \equiv 4 \mod 32 \qquad valid(\mu.\texttt{gas}, 3, |\mathbf{s}| + 1) \qquad |\iota.\texttt{input}| < a + 31 \\ \exists i \in \{0, \ldots, k-1\} : \iota.\texttt{input}[a + i] \notin \mathbb{N}_8 \qquad \mathbf{cond} = \mu.\texttt{cond}\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to EXC(\mathbf{cond}) :: S}$$

$$\frac{\omega_{\mu,\iota} = \texttt{CALLDATALOAD} \qquad \mu.\mathbf{s} = a :: \mathbf{s} \qquad (a \neq 0 \wedge a \not\equiv 4 \mod 32) \qquad \mathbf{cond} = \mu.\texttt{cond}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to EXC(\mathbf{cond}) :: S}$$

$$\frac{\omega_{\mu,\iota} = \texttt{CALLDATALOAD} \quad (\neg valid(\mu.\texttt{gas}, 3, |\mu.\texttt{s}| + 1) \vee |\mu.\texttt{s}| < 1) \quad \textbf{cond} = \mu.\texttt{cond}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to EXC(\textbf{cond}) :: S}$$

`CALLDATACOPY` accepts numerical positions and size and copies the desired portion of the input to the memory, whether it has symbols or not. The semantic rules are straightforward and we omit them for brevity. `CODECOPY` accepts numerical positions and size and copies the currently executing code, which is an array of numbers. `EXTCODECOPY` is analogous, but it also receives an address. The address may be a symbol but the code stored under that symbol needs to be numeric. Thus, these two opcodes only suffer slight modifications.

Since we are only interested in the small-step execution of transactions, `GASPRICE` may return a symbol.

`RETURNDATACOPY` does not allow symbolic arguments since they are positions and a size. The rules for `RETURNDATACOPY` and `RETURNDATASIZE` had not been formalized prior to this work, but they are very similar to those of `CALLDATACOPY` and `CALLDATASIZE`.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{RETURNDATACOPY} \\ \mu.\texttt{s} = pos_{\texttt{m}} :: pos_{\texttt{d}} :: size :: \textbf{s} \qquad aw = M(\mu.\texttt{i}, pos_{\texttt{m}}, size) \\ c = C_{mem}(\mu.\texttt{i}, aw) + 3 + 3 \cdot \left\lceil \frac{size}{32} \right\rceil \qquad valid(\mu.\texttt{gas}, c, |\textbf{s}|) \\ k = |\mu.\texttt{rd}| - pos_{\texttt{d}} < 0 \ ? \ 0 : min(|\mu.\texttt{rd}| - pos_{\texttt{d}} - size) \\ d' = \mu.\texttt{rd}[pos_{\texttt{d}}, pos_{\texttt{d}} + k - 1] \qquad d = d' \cdot 0^{8 \cdot (size - k)} \\ \mu' = \mu[\texttt{gas} \ -= c][\texttt{pc} \ += 1][\texttt{m} \to \mu.\texttt{m}[[pos_{\texttt{m}}, pos_{\texttt{m}} + size - 1] \to d][\texttt{i} \to aw][\texttt{s} \to \textbf{s}]\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{RETURNDATACOPY} \\ \mu.\texttt{s} = pos_{\texttt{m}} :: pos_{\texttt{d}} :: size :: \textbf{s} \qquad aw = M(\mu.\texttt{i}, pos_{\texttt{m}}, size) \\ c = C_{mem}(\mu.\texttt{i}, aw) + 3 + 3 \cdot \left\lceil \frac{size}{32} \right\rceil \quad \neg(valid(\mu.\texttt{gas}, c, |\textbf{s}|)) \quad \textbf{cond} = \mu.\texttt{cond}\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to EXC(\textbf{cond}) :: S}$$

$$\frac{\omega_{\mu,\iota} = \texttt{RETURNDATACOPY} \quad |\mu.\texttt{s}| < 2 \quad \textbf{cond} = \mu.\texttt{cond}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to EXC(\textbf{cond}) :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{RETURNDATASIZE} \\ valid(\mu.\texttt{gas}, 2, |\mu.\texttt{s}| + 1) \quad \mu' = \mu[\texttt{gas} \ -= 2][\texttt{pc} \ += 1][\texttt{s} \to |\mu.\texttt{rd}| :: \textbf{s}]\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \texttt{RETURNDATASIZE} \quad \neg(valid(\mu.\texttt{gas}, 2, |\mu.\texttt{s}| + 1)) \quad \textbf{cond} = \mu.\texttt{cond}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to EXC(\textbf{cond}) :: S}$$

A.6 Block information

The operations `COINBASE`, `TIMESTAMP`, `NUMBER`, `DIFFICULTY`, and `GASLIMIT` may retrieve symbols and their semantics are as expected.

A.7 Stack, memory, storage and flow operations

POP, JUMP, PC, MSIZE, GAS and JUMPDEST semantic rules do not suffer any relevant modification since none of them deals with symbolic arguments.

MSTORE receives two arguments from the stack, a and b, and saves b in the memory as a word: b is written in the 32 bytes from position a to position $a + 31$. a must be a number; b may be a symbol. If b is a symbol it is stored in a different way: 32 copies of the symbol are written in the desired portion of the memory.

$$\frac{\begin{array}{ccc}\omega_{\mu,\iota} = \texttt{MSTORE} & c = C_{mem}(aw) - C_{mem}(\mu.\texttt{i}) + 3 & valid(\mu.\texttt{gas}, c, |\mathbf{s}|) \\ aw = M(\mu.\texttt{i}, a, 32) & \mu.\mathbf{s} = a :: \mathbf{b} :: \mathbf{s} & \mathbf{b} \notin \mathbb{N}_{256} \\ \multicolumn{3}{c}{\mu' = \mu[\texttt{gas}- = c][\texttt{pc}+ = 1][\texttt{m} \to \mu.\texttt{m}[a \to \mathbf{b}] \dots [a + 31 \to \mathbf{b}]][\texttt{i} \to aw][\texttt{s} \to \mathbf{s}]}\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\begin{array}{ccc}\omega_{\mu,\iota} = \texttt{MSTORE} & c = C_{mem}(aw) - C_{mem}(\mu.\texttt{i}) + 3 & \\ aw = M(\mu.\texttt{i}, a, 32) & \neg valid(\mu.\texttt{gas}, c, |\mu.\mathbf{s}|) & \mathbf{cond} = \mu.\texttt{cond}\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to EXC(\mathbf{cond}) :: S}$$

MSTORE8 stores a single byte. It should be noted that, if a symbol is stored through MSTORE8, it will not be possible to retrieve it unless there are 31 equal symbols stored in contiguous positions of the memory.

$$\frac{\begin{array}{ccc}\omega_{\mu,\iota} = \texttt{MSTORE8} & c = C_{mem}(aw) - C_{mem}(\mu.\texttt{i}) + 3 & \\ valid(\mu.\texttt{gas}, c, |\mathbf{s}|) & aw = M(\mu.\texttt{i}, a, 1) & \mu.\mathbf{s} = a :: \mathbf{b} :: \mathbf{s} \\ \multicolumn{3}{c}{\mu' = \mu[\texttt{gas}- = c][\texttt{pc}+ = 1][\texttt{m} \to \mu.\texttt{m}[a \to \mathbf{b} \mod 256]][\texttt{i} \to aw][\texttt{s} \to \mathbf{s}]}\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\begin{array}{ccc}\omega_{\mu,\iota} = \texttt{MSTORE8} & c = C_{mem}(aw) - C_{mem}(\mu.\texttt{i}) + 3 & \\ aw = M(\mu.\texttt{i}, a, 1) & \neg valid(\mu.\texttt{gas}, c, |\mu.\mathbf{s}|) & \mathbf{cond} = \mu.\texttt{cond}\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to EXC(\mathbf{cond}) :: S}$$

MLOAD receives a numerical position a and pushes the element in that position of the memory to the stack. If that element is symbolic, it checks if there are 32 copies of it starting in position a. If there are, it pushes that symbol to the stack; otherwise it raises an exception.

$$\frac{\begin{array}{ccccc}\omega_{\mu,\iota} = \texttt{MLOAD} & c = C_{mem}(aw) - C_{mem}(\mu.\texttt{i}) + 3 & & & \mu.\mathbf{s} = a :: \mathbf{s} \\ valid(\mu.\texttt{gas}, c, |\mathbf{s}| + 1) & aw = M(\mu.\texttt{i}, a, 32) & & & \mu.\texttt{m}[a] = \dots = \mu.\texttt{m}[a + 31] = \mathbf{v} \\ \multicolumn{5}{c}{\mu' = \mu[\texttt{gas}- = c][\texttt{pc}+ = 1][\texttt{i} \to aw][\texttt{s} \to \mathbf{v} :: \mathbf{s}]}\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\begin{array}{ccc}\omega_{\mu,\iota} = \texttt{MLOAD} & aw = M(\mu.\texttt{i}, a, 32) & \mu.\mathbf{s} = a :: \mathbf{s} \\ \mathbf{cond} = \mu.\texttt{cond} & c = C_{mem}(aw) - C_{mem}(\mu.\texttt{i}) + 3 & \\ \multicolumn{3}{c}{((\exists i \in \{0, \dots, 31\} \;\; \mu.\texttt{m}[a + i] \notin \mathbb{N}_8 \wedge \neg(\mu.\texttt{m}[a] = \dots = \mu.\texttt{m}[a + 31])) \vee \neg valid(\mu.\texttt{gas}, c, |s| + 1))}\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to EXC(\mathbf{cond}) :: S}$$

$$\frac{\omega_{\mu,\iota} = \texttt{MLOAD} \quad |\mu.\mathbf{s}| < 1 \quad \mathbf{cond} = \mu.\texttt{cond}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to EXC(\mathbf{cond}) :: S}$$

The storage is handled in a different way since it stores words instead of bytes. It is possible to store a symbolic element in a symbolic position. There are four possible branches since the refund balance and the gas cost depend on whether the storage contents or the position we are trying to write to are 0 or not and whether the new content is 0 or not. We start with the case where there is enough gas for every possibility.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{SSTORE} \quad valid(\mu.\texttt{gas}, 20000, |\mathbf{s}|) \quad \mu.\mathbf{s} = \mathbf{a} :: \mathbf{b} :: \mathbf{s} \quad \mathbf{x} = (\sigma(\iota.\texttt{actor}).\texttt{stor})(\mathbf{a}) \\ \mu'_1 = \mu[\texttt{gas}- = 5000][\texttt{pc}+ = 1][\mathbf{s} \to \mathbf{s}][\texttt{cond} \to \mu.\texttt{cond} \wedge \mathbf{x} = 0 \wedge \mathbf{b} = 0] \\ \mu'_2 = \mu[\texttt{gas}- = 20000][\texttt{pc}+ = 1][\mathbf{s} \to \mathbf{s}][\texttt{cond} \to \mu.\texttt{cond} \wedge \mathbf{x} = 0 \wedge \mathbf{b} \neq 0] \\ \mu'_3 = \mu[\texttt{gas}- = 5000][\texttt{pc}+ = 1][\mathbf{s} \to \mathbf{s}][\texttt{cond} \to \mu.\texttt{cond} \wedge \mathbf{x} \neq 0 \wedge \mathbf{b} = 0] \\ \mu'_4 = \mu[\texttt{gas}- = 5000][\texttt{pc}+ = 1][\mathbf{s} \to \mathbf{s}][\texttt{cond} \to \mu.\texttt{cond} \wedge \mathbf{x} \neq 0 \wedge \mathbf{b} \neq 0] \\ \sigma' = \sigma\langle\iota.\texttt{actor} \to \iota.\texttt{actor}[\texttt{stor} \to \sigma(\iota.\texttt{actor}).\texttt{stor}[\mathbf{a} \to \mathbf{b}]]\rangle \quad \eta'_3 = \eta[\texttt{balr}+ = 15000]\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to \begin{array}{l}(\mu'_1, \iota, \sigma', \eta) :: S \\ (\mu'_2, \iota, \sigma', \eta) :: S \\ (\mu'_3, \iota, \sigma', \eta'_3) :: S \\ (\mu'_4, \iota, \sigma', \eta) :: S\end{array}}$$

Since three of the possibilities require 5000 units of gas and the other uses 20000, if the available gas is between these two values we need to return that information inside the conditions of the exceptions.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{SSTORE} \quad \neg valid(\mu.\texttt{gas}, 20000, |\mathbf{s}|) \quad valid(\mu.\texttt{gas}, 5000, |\mathbf{s}|) \\ \mu.\mathbf{s} = \mathbf{a} :: \mathbf{b} :: \mathbf{s} \quad \mathbf{x} = (\sigma(\iota.\texttt{actor}).\texttt{stor})(\mathbf{a}) \quad \mathbf{cond} = \mu.\texttt{cond} \\ \mu'_1 = \mu[\texttt{gas}- = 5000][\texttt{pc}+ = 1][\mathbf{s} \to \mathbf{s}][\texttt{cond} \to \mu.\texttt{cond} \wedge \mathbf{x} = 0 \wedge \mathbf{b} = 0] \\ \mu'_3 = \mu[\texttt{gas}- = 5000][\texttt{pc}+ = 1][\mathbf{s} \to \mathbf{s}][\texttt{cond} \to \mu.\texttt{cond} \wedge \mathbf{x} \neq 0 \wedge \mathbf{b} = 0] \\ \mu'_4 = \mu[\texttt{gas}- = 5000][\texttt{pc}+ = 1][\mathbf{s} \to \mathbf{s}][\texttt{cond} \to \mu.\texttt{cond} \wedge \mathbf{x} \neq 0 \wedge \mathbf{b} \neq 0] \\ \sigma' = \sigma\langle\iota.\texttt{actor} \to \iota.\texttt{actor}[\texttt{stor} \to \sigma(\iota.\texttt{actor}).\texttt{stor}[\mathbf{a} \to \mathbf{b}]]\rangle \quad \eta' = \eta[\texttt{balr}+ = 15000]\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to \begin{array}{l}(\mu'_1, \iota, \sigma', \eta) :: S \\ EXC(\mathbf{cond} \wedge \mathbf{x} = 0 \wedge \mathbf{b} \neq 0) :: S \\ (\mu'_3, \iota, \sigma', \eta') :: S \\ (\mu'_4, \iota, \sigma', \eta) :: S\end{array}}$$

$$\frac{\omega_{\mu,\iota} = \texttt{SSTORE} \quad (\neg valid(\mu.\texttt{gas}, 5000, |s|) \vee |\mu.\mathbf{s}| < 2) \quad \mathbf{cond} = \mu.\texttt{cond}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to EXC(\mathbf{cond}) :: S}$$

It is also possible to retrieve an element from a symbolic position. If the storage of the contract has something in that symbolic position, it returns it, otherwise it returns 0.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{SLOAD} \quad valid(\mu.\texttt{gas}, 200, |\mathbf{s}| + 1) \quad \mathbf{x} = (\sigma(\iota.\texttt{actor}).\texttt{stor})(\mathbf{a}) \\ \mu.\mathbf{s} = \mathbf{a} :: \mathbf{s} \quad \mu' = \mu[\texttt{gas}- = 200][\texttt{pc}+ = 1][\mathbf{s} \to \mathbf{x} :: s]\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \texttt{SLOAD} \quad (\neg valid(\mu.\texttt{gas}, 200, |\mu.\mathbf{s}| + 1) \vee |\mu.\mathbf{s}| < 1) \quad \mathbf{cond} = \mu.\texttt{cond}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to EXC(\mathbf{cond}) :: S}$$

The JUMPI instruction may cause the current call stack to split. The first argument cannot be symbolic because it represents a position of the code.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{JUMPI} \quad valid(\mu.\texttt{gas}, 10, |\mathbf{s}|) \quad \mu.\mathbf{s} = i :: \mathbf{b} :: \mathbf{s} \quad i \in D(\iota.\texttt{code}) \\ \mu'_1 = \mu[\texttt{pc}+ = 1][\mathbf{s} \to \mathbf{s}][\texttt{gas}- = 10][\texttt{cond} \to \mu.\texttt{cond} \wedge \mathbf{b} = 0] \\ \mu'_2 = \mu[\texttt{pc} \to i][\mathbf{s} \to \mathbf{s}][\texttt{gas}- = 10][\texttt{cond} \to \mu.\texttt{cond} \wedge \mathbf{b} \neq 0]\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to \begin{array}{l}(\mu'_1, \iota, \sigma, \eta) :: (\mu, \iota, \sigma, \eta) :: S \\ (\mu'_2, \iota, \sigma, \eta) :: (\mu, \iota, \sigma, \eta) :: S\end{array}}$$

A.8 Push, dup and swap operations

A push operation pushes elements from the bytecode to the stack. Since the bytecode does not contain symbols, it would not make sense to push a symbol to the stack. The semantics of the 32 push instructions remain unchanged. A dup or a swap operation may duplicate or exchange numeric or symbolic elements; the rules are straightforward to derive.

A.9 Logging operations

The memory contents accessed by the logging operations can contain symbols, but the positions and the sizes must be numeric. The execution stack elements taken by some of these operations may be symbolic. Thus, the semantic rules are very similar to the original ones.

A.10 System operations

Contract creation. The first rule concerns the case where the newly generated address corresponds to an account that does not exist yet.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{CREATE} \\ \mu.\mathbf{s} = \mathbf{va} :: io :: is :: \mathbf{s} \quad aw = M(\mu.\texttt{i}, io, is) \quad c = C_{mem}(\mu.\texttt{i}, aw) + 32000 \\ valid(\mu.\texttt{gas}, c, |\mathbf{s}| + 1) \quad \rho = newAddress(\iota.\texttt{actor}, \sigma(\iota.\texttt{actor}).\texttt{n}) \quad \sigma(\rho) = \bot \\ |S| + 1 \leq 1024 \quad \mu' = (L(\mu.\texttt{gas} - c), 0, \lambda x.0, 0, \epsilon, \epsilon, \mu.\texttt{cond}) \\ i = \mu.\texttt{m}[io, io + is - 1] \quad \iota' = (\rho, i, \iota.\texttt{actor}, \mathbf{va}, \epsilon) \\ \sigma' = \sigma\langle\rho \to (0, \mathbf{va}, \epsilon, \lambda x.0)\rangle\langle\iota.\texttt{actor} \to \sigma(\iota.\texttt{actor})[\texttt{b}- = \mathbf{va}][\texttt{n}+ = 1]\rangle\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to \begin{array}{l}(\mu', \iota', \sigma', \eta) :: (\mu, \iota, \sigma, \eta) :: S \\ EXC(\mu.\texttt{cond} \wedge \mathbf{va} > \sigma(\iota.\texttt{actor}).\mathbf{b}) :: (\mu, \iota, \sigma, \eta) :: S\end{array}}$$

The modifications for the case where the new address exists are the same as the original semantics specification.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{CREATE} \\ \mu.\mathbf{s} = \mathbf{va} :: io :: is :: \mathbf{s} \qquad aw = M(\mu.\mathtt{i}, io, is) \qquad c = C_{mem}(\mu.\mathtt{i}, aw) + 32000 \\ valid(\mu.\mathtt{gas}, c, |\mathbf{s}| + 1) \qquad \rho = newAddress(\iota.\mathtt{actor}, \sigma(\iota.\mathtt{actor}).\mathtt{n}) \qquad \sigma(\rho) \neq \bot \\ |S| + 1 \leq 1024 \qquad \mathbf{ba} = \sigma(\rho).\mathtt{b} + \mathbf{va} \qquad \mu' = (L(\mu.\mathtt{gas} - c), 0, \lambda x.0, 0, \epsilon, \epsilon, \mu.\mathtt{cond}) \\ i = \mu.\mathtt{m}[io, io + is - 1] \qquad \iota' = (\rho, i, \iota.\mathtt{actor}, \mathbf{va}, \epsilon) \\ \sigma' = \sigma\langle\rho \to (0, \mathbf{ba}, \epsilon, \lambda x.0)\rangle\langle\iota.\mathtt{actor} \to \sigma(\iota.\mathtt{actor})[\mathtt{b}- = \mathbf{va}][\mathtt{n}+ = 1]\rangle\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to \begin{array}{l}(\mu', \iota', \sigma', \eta) :: (\mu, \iota, \sigma, \eta) :: S \\ EXC(\mu.\mathtt{cond} \wedge \mathbf{va} > \sigma(\iota.\mathtt{actor}).\mathbf{b}) :: (\mu, \iota, \sigma, \eta) :: S\end{array}}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{CREATE} \\ \mu.\mathbf{s} = \mathbf{va} :: io :: is :: \mathbf{s} \qquad aw = M(\mu.\mathtt{i}, io, is) \qquad c = C_{mem}(\mu.\mathtt{i}, aw) + 32000 \\ valid(\mu.\mathtt{gas}, c, |\mathbf{s}| + 1) \qquad |S| + 1 > 1024 \qquad cond = \mu.\mathtt{cond}\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to EXC(cond) :: (\mu, \iota, \sigma, \eta) :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{CREATE} \qquad \mu.\mathbf{s} = \mathbf{va} :: io :: is :: \mathbf{s} \\ aw = M(\mu.\mathtt{i}, io, is) \qquad c = C_{mem}(\mu.\mathtt{i}, aw) + 32000 \qquad \neg valid(\mu.\mathtt{gas}, c, |\mathbf{s}| + 1)\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to EXC(\mu.\mathtt{cond}) :: S}$$

$$\frac{\omega_{\mu,\iota} = \texttt{CREATE} \qquad |\mu.\mathbf{s}| < 3}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to EXC(\mu.\mathtt{cond}) :: S}$$

We correct a mistake in the semantics by Grishchenko et al. in [6]: when returning successfully from a contract creation, we need to subtract the gas that was given to it.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{CREATE} \\ \mu.\mathbf{s} = \mathbf{va} :: io :: is :: \mathbf{s} \qquad aw = M(\mu.\mathtt{i}, io, is) \qquad c = C_{mem}(\mu.\mathtt{i}, aw) + 32000 \\ c_{final} = 200 \cdot |d| \qquad gas \geq c_{final} \qquad \rho = newAddress(\iota.\mathtt{actor}, \sigma(\iota.\mathtt{actor}).\mathtt{n}) \\ \sigma'' = \sigma'\langle\rho \to \sigma'(\rho)[\mathtt{code} \to d]\rangle \\ \mu' = \mu[\mathtt{gas} \ += gas - L(\mu.\mathtt{gas} - c) - c - c_{final}][\mathtt{pc} \ += 1][\mathtt{i} \to aw][\mathtt{s} \to \rho :: \mathbf{s}][\mathtt{rd} \to d][\mathtt{cond} \to \mathbf{cond}]\end{array}}{\Gamma \models HALT(\sigma', g, d, \eta', \mathbf{cond}) :: (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma'', \eta') :: S}$$

$$\frac{\omega_{\mu,\iota} = \texttt{CREATE} \qquad c_{final} = 200 \cdot |d| \qquad gas < c_{final}}{\Gamma \models HALT(\sigma', g, d, \eta', \mathbf{cond}) :: (\mu, \iota, \sigma, \eta) :: S \to EXC(\mathbf{cond}) :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{CREATE} \\ \mu.\mathbf{s} = \mathbf{va} :: io :: is :: \mathbf{s} \qquad aw = M(\mu.\mathtt{i}, io, is) \qquad c = C_{mem}(\mu.\mathtt{i}, aw) + 32000 \\ c_{final} = 200 \cdot |d| \qquad gas \geq c_{final} \\ \mu' = \mu[\mathtt{gas} \ += gas - L(\mu.\mathtt{gas} - c) - c - c_{final}][\mathtt{pc} \ += 1][\mathtt{i} \to aw][\mathtt{s} \to 0 :: \mathbf{s}][\mathtt{rd} \to d][\mathtt{cond} \to \mathbf{cond}]\end{array}}{\Gamma \models REV(g, d, \mathbf{cond}) :: (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

If the contract creation ended unsuccessfully, the return data is empty.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{CREATE} \qquad \mu.\mathbf{s} = \mathbf{va} :: io :: is :: \mathbf{s} \qquad aw = M(\mu.\mathtt{i}, io, is) \qquad c = C_{mem}(\mu.\mathtt{i}, aw) + 32000 \\ \mu' = \mu[\mathtt{gas} \ -= c][\mathtt{pc} \ += 1][\mathtt{i} \to aw][\mathtt{s} \to 0 :: \mathbf{s}][\mathtt{rd} \to \epsilon][\mathtt{cond} \to \mathbf{cond}]\end{array}}{\Gamma \models EXC(\mathbf{cond}) :: (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

Call operation. In the first case that we consider, the value is unknown and the called account exists. The success of the operation depends on the relationship between the balance of the sender and the transferred value, and on the existence of enough gas. The gas cost depends on the value, the flag and the gas specified by the sender. Since we do not allow the flag or the gas to be symbolic, we only need to care about the value va. Therefore, there are only three possible cases: $va = 0$, $0 < va \leq \sigma(\iota.\texttt{actor.b})$ and $va > \sigma(\iota.\texttt{actor.b})$. The third case triggers an exception; the first and the second cases need to be considered separately because the gas calculations are different. In the second case we consider $va = 1$ in the gas calculations because the result is the same with every non-zero value. The input may contain symbolic elements. Notice that in order to call a symbolic address **to** it is necessary that either the symbol $\mathbf{to} \mod 2^{160}$ is stored on the global state σ or that $isAddress(\mathbf{to})$ is implied by the condition $\mu.\texttt{cond}$.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{CALL} \qquad \mu.\mathbf{s} = g :: \mathbf{to} :: \mathbf{va} :: io :: is :: oo :: os :: \mathbf{s} \qquad |S| + 1 \leq 1024 \\ \mathbf{to}_a = \mathbf{to} \mod 2^{160} \quad \sigma(\mathbf{to}_a) \neq \bot \quad aw = M(M(\mu.\mathtt{i}, io, is), oo, os) \quad \mathbf{d} = \mu.\mathtt{m}[io, io + is - 1] \\ c_{call_1} = C_{gascap}(0, 1, g, \mu.\texttt{gas}) \qquad c_{call_2} = C_{gascap}(1, 1, g, \mu.\texttt{gas}) \\ c_1 = C_{base}(0, 1) + C_{mem}(\mu.\mathtt{i}) + c_{call_1} \qquad c_2 = C_{base}(1, 1) + C_{mem}(\mu.\mathtt{i}) + c_{call_2} \\ valid(\mu.\texttt{gas}, c_1, |\mathbf{s}| + 1) \qquad valid(\mu.\texttt{gas}, c_2, |\mathbf{s}| + 1) \\ \sigma' = \sigma\langle \mathbf{to}_a \to \sigma(\mathbf{to}_a)[\mathtt{b} + = \mathbf{va}]\rangle\langle \iota.\texttt{actor} \to \sigma(\iota.\texttt{actor})[\mathtt{b} - = \mathbf{va}]\rangle \\ \mu'_1 = (c_{call_1}, 0, \lambda x.0, 0, \epsilon, \mu.\mathtt{rd}, \mu.\texttt{cond} \wedge \mathbf{va} = 0) \\ \mu'_2 = (c_{call_2}, 0, \lambda x.0, 0, \epsilon, \mu.\mathtt{rd}, \mu.\texttt{cond} \wedge 0 < \mathbf{va} \leq \sigma(\iota.\texttt{actor}).\mathtt{b}) \\ \iota'_1 = \iota[\texttt{actor} \to to_a][\texttt{input} \to \mathbf{d}][\texttt{sender} \to \iota.\texttt{actor}][\texttt{value} \to 0][\texttt{code} \to \sigma(to_a).\texttt{code}] \\ \iota'_2 = \iota[\texttt{actor} \to to_a][\texttt{input} \to \mathbf{d}][\texttt{sender} \to \iota.\texttt{actor}][\texttt{value} \to \mathbf{va}][\texttt{code} \to \sigma(to_a).\texttt{code}]\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to \begin{array}{l}(\mu'_1, \iota'_1, \sigma, \eta) :: (\mu, \iota, \sigma, \eta) :: S \\ (\mu'_2, \iota'_2, \sigma', \eta) :: (\mu, \iota, \sigma, \eta) :: S \\ EXC(\mu.\texttt{cond} \wedge \mathbf{va} > \sigma(\iota.\texttt{actor}).\mathtt{b})) :: (\mu, \iota, \sigma, \eta) :: S\end{array}}$$

When the called account does not exist it is necessary to create it.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{CALL} \qquad \mu.\mathbf{s} = g :: \mathbf{to} :: \mathbf{va} :: io :: is :: oo :: os :: \mathbf{s} \qquad |S| + 1 \leq 1024 \\ \mathbf{to}_a = \mathbf{to} \mod 2^{160} \quad \sigma(\mathbf{to}_a) = \bot \quad aw = M(M(\mu.\mathtt{i}, io, is), oo, os) \quad \mathbf{d} = \mu.\mathtt{m}[io, io + is - 1] \\ c_{call_1} = C_{gascap}(0, 0, g, \mu.\texttt{gas}) \qquad c_{call_2} = C_{gascap}(1, 0, g, \mu.\texttt{gas}) \\ c_1 = C_{base}(0, 0) + C_{mem}(\mu.\mathtt{i}) + c_{call_1} \qquad c_2 = C_{base}(1, 0) + C_{mem}(\mu.\mathtt{i}) + c_{call_2} \\ valid(\mu.\texttt{gas}, c_1, |\mathbf{s}| + 1) \qquad valid(\mu.\texttt{gas}, c_2, |\mathbf{s}| + 1) \\ \sigma' = \sigma\langle \mathbf{to}_a \to (0, \mathbf{va}, \epsilon, \lambda x.0)\rangle\langle \iota.\texttt{actor} \to \sigma(\iota.\texttt{actor})[\mathtt{b} - = \mathbf{va}]\rangle \\ \mu'_1 = (c_{call_1}, 0, \lambda x.0, 0, \epsilon, \mu.\mathtt{rd}, \mu.\texttt{cond} \wedge \mathbf{va} = 0) \\ \mu'_2 = (c_{call_2}, 0, \lambda x.0, 0, \epsilon, \mu.\mathtt{rd}, \mu.\texttt{cond} \wedge 0 < \mathbf{va} \leq \sigma(\iota.\texttt{actor}).\mathtt{b}) \\ \iota'_1 = \iota[\texttt{actor} \to to_a][\texttt{input} \to \mathbf{d}][\texttt{sender} \to \iota.\texttt{actor}][\texttt{value} \to 0][\texttt{code} \to \epsilon] \\ \iota'_2 = \iota[\texttt{actor} \to to_a][\texttt{input} \to \mathbf{d}][\texttt{sender} \to \iota.\texttt{actor}][\texttt{value} \to \mathbf{va}][\texttt{code} \to \epsilon]\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to \begin{array}{l}(\mu'_1, \iota'_1, \sigma, \eta) :: (\mu, \iota, \sigma, \eta) :: S \\ (\mu'_2, \iota'_2, \sigma', \eta) :: (\mu, \iota, \sigma, \eta) :: S \\ EXC(\mu.\texttt{cond} \wedge \mathbf{va} > \sigma(\iota.\texttt{actor}).\mathtt{b})) :: (\mu, \iota, \sigma, \eta) :: S\end{array}}$$

The cost of the `CALL` operation is higher when the value is greater than 0. Therefore we need to consider the case where there is enough gas to `CALL` a contract without sending ether but there is not enough gas to send any amount. We split the analysis in two cases: the account exists and the account does not exist.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{CALL} \qquad \mu.\mathbf{s} = g :: \mathbf{to} :: \mathbf{va} :: io :: is :: oo :: os :: \mathbf{s} \qquad |S| + 1 \leq 1024 \\ \mathbf{to}_a = \mathbf{to} \mod 2^{160} \qquad \sigma(\mathbf{to}_a) \neq \bot \qquad aw = M(M(\mu.\mathtt{i}, io, is), oo, os) \qquad \mathbf{d} = \mu.\mathtt{m}[io, io + is - 1] \\ c_{call_1} = C_{gascap}(0, 1, g, \mu.\mathtt{gas}) \qquad c_{call_2} = C_{gascap}(1, 1, g, \mu.\mathtt{gas}) \\ c_1 = C_{base}(0, 1) + C_{mem}(\mu.\mathtt{i}) + c_{call_1} \qquad c_2 = C_{base}(1, 1) + C_{mem}(\mu.\mathtt{i}) + c_{call_2} \\ valid(\mu.\mathtt{gas}, c_1, |\mathbf{s}| + 1) \qquad \neg valid(\mu.\mathtt{gas}, c_2, |\mathbf{s}| + 1) \\ \mu' = (c_{call_1}, 0, \lambda x.0, 0, \epsilon, \mu.\mathtt{rd}, \mu.\mathtt{cond} \wedge \mathbf{va} = 0) \\ \iota' = \iota[\mathtt{actor} \rightarrow to_a][\mathtt{input} \rightarrow \mathbf{d}][\mathtt{sender} \rightarrow \iota.\mathtt{actor}][\mathtt{value} \rightarrow 0][\mathtt{code} \rightarrow \sigma(to_a).\mathtt{code}]\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \rightarrow \begin{array}{l}(\mu', \iota', \sigma, \eta) :: (\mu, \iota, \sigma, \eta) :: S \\ EXC(\mu.\mathtt{cond} \wedge 0 < \mathbf{va} \leq \sigma(\iota.\mathtt{actor}).\mathtt{b})) :: S \\ EXC(\mu.\mathtt{cond} \wedge \mathbf{va} > \sigma(\iota.\mathtt{actor}).\mathtt{b})) :: (\mu, \iota, \sigma, \eta) :: S\end{array}}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{CALL} \qquad \mu.\mathbf{s} = g :: \mathbf{to} :: \mathbf{va} :: io :: is :: oo :: os :: \mathbf{s} \qquad |S| + 1 \leq 1024 \\ \mathbf{to}_a = \mathbf{to} \mod 2^{160} \qquad \sigma(\mathbf{to}_a) = \bot \qquad aw = M(M(\mu.\mathtt{i}, io, is), oo, os) \qquad \mathbf{d} = \mu.\mathtt{m}[io, io + is - 1] \\ c_{call_1} = C_{gascap}(0, 0, g, \mu.\mathtt{gas}) \qquad c_{call_2} = C_{gascap}(1, 0, g, \mu.\mathtt{gas}) \\ c_1 = C_{base}(0, 0) + C_{mem}(\mu.\mathtt{i}) + c_{call_1} \qquad c_2 = C_{base}(1, 0) + C_{mem}(\mu.\mathtt{i}) + c_{call_2} \\ valid(\mu.\mathtt{gas}, c_1, |\mathbf{s}| + 1) \qquad \neg valid(\mu.\mathtt{gas}, c_2, |\mathbf{s}| + 1) \\ \mu' = (c_{call_1}, 0, \lambda x.0, 0, \epsilon, \mu.\mathtt{rd}, \mu.\mathtt{cond} \wedge \mathbf{va} = 0) \\ \iota' = \iota[\mathtt{actor} \rightarrow to_a][\mathtt{input} \rightarrow \mathbf{d}][\mathtt{sender} \rightarrow \iota.\mathtt{actor}][\mathtt{value} \rightarrow 0][\mathtt{code} \rightarrow \epsilon]\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \rightarrow \begin{array}{l}(\mu', \iota', \sigma, \eta) :: (\mu, \iota, \sigma, \eta) :: S \\ EXC(\mu.\mathtt{cond} \wedge 0 < \mathbf{va} \leq \sigma(\iota.\mathtt{actor}).\mathtt{b})) :: S \\ EXC(\mu.\mathtt{cond} \wedge \mathbf{va} > \sigma(\iota.\mathtt{actor}).\mathtt{b})) :: (\mu, \iota, \sigma, \eta) :: S\end{array}}$$

Finally, we need to consider the case where there is not enough gas for any `CALL`. Since the cost with posiitve value is greater than the cost with value equal to zero, it is enough to consider as a premise that there is not enough gas for a call without transference of ether.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{CALL} \qquad \mu.\mathbf{s} = g :: \mathbf{to} :: \mathbf{va} :: io :: is :: oo :: os :: \mathbf{s} \qquad |S| + 1 \leq 1024 \\ \mathbf{to}_a = \mathbf{to} \mod 2^{160} \qquad aw = M(M(\mu.\mathtt{i}, io, is), oo, os) \\ flag = (\sigma(\mathbf{to}_a) = \bot)\ ?\ 0\ :\ 1 \qquad c_{call} = C_{gascap}(0, flag, g, \mu.\mathtt{gas}) \\ c = C_{base}(0, flag) + C_{mem}(\mu.\mathtt{i}) + c_{call} \qquad \neg valid(\mu.\mathtt{gas}, c, |\mathbf{s}| + 1)\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \rightarrow \begin{array}{l}EXC(\mu.\mathtt{cond} \wedge \mathbf{va} = 0) :: (\mu, \iota, \sigma, \eta) :: S \\ EXC(\mu.\mathtt{cond} \wedge 0 < \mathbf{va} \leq \sigma(\iota.\mathtt{actor}).\mathtt{b})) :: S \\ EXC(\mu.\mathtt{cond} \wedge \mathbf{va} > \sigma(\iota.\mathtt{actor}).\mathtt{b})) :: (\mu, \iota, \sigma, \eta) :: S\end{array}}$$

The called account exists and there are more elements in the call stack than allowed.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{CALL} \qquad \mu.\mathbf{s} = g :: \mathbf{to} :: \mathbf{va} :: io :: is :: oo :: os :: \mathbf{s} \qquad |S| + 1 > 1024 \\ \mathbf{to}_a = \mathbf{to} \mod 2^{160} \qquad \sigma(\mathbf{to}_a) \neq \bot \qquad aw = M(M(\mu.\mathtt{i}, io, is), oo, os) \\ c_{call_1} = C_{gascap}(0, 1, g, \mu.\mathtt{gas}) \qquad c_{call_2} = C_{gascap}(1, 1, g, \mu.\mathtt{gas}) \\ c_1 = C_{base}(0, 1) + C_{mem}(\mu.\mathtt{i}) + c_{call_1} \qquad c_2 = C_{base}(1, 1) + C_{mem}(\mu.\mathtt{i}) + c_{call_2} \\ valid(\mu.\mathtt{gas}, c_1, |\mathbf{s}| + 1) \qquad valid(\mu.\mathtt{gas}, c_2, |\mathbf{s}| + 1)\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \rightarrow \begin{array}{l}EXC(\mu.\mathtt{cond} \wedge \mathbf{va} = 0) :: (\mu, \iota, \sigma, \eta) :: S \\ EXC(\mu.\mathtt{cond} \wedge 0 < \mathbf{va} \leq \sigma(\iota.\mathtt{actor}).\mathtt{b}) :: (\mu, \iota, \sigma, \eta) :: S \\ EXC(\mu.\mathtt{cond} \wedge \mathbf{va} > \sigma(\iota.\mathtt{actor}).\mathtt{b})) :: (\mu, \iota, \sigma, \eta) :: S\end{array}}$$

The called account does not exist and the operation causes more elements to be in the call stack than what is permitted.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{CALL} \qquad \mu.\mathbf{s} = g :: \mathbf{to} :: \mathbf{va} :: io :: is :: oo :: os :: \mathbf{s} \qquad |S| + 1 > 1024 \\ \mathbf{to}_a = \mathbf{to} \mod 2^{160} \qquad \sigma(\mathbf{to}_a) = \bot \qquad aw = M(M(\mu.\texttt{i}, io, is), oo, os) \\ c_{call_1} = C_{gascap}(0, 0, g, \mu.\texttt{gas}) \qquad c_{call_2} = C_{gascap}(1, 0, g, \mu.\texttt{gas}) \\ c_1 = C_{base}(0, 0) + C_{mem}(\mu.\texttt{i}) + c_{call_1} \qquad c_2 = C_{base}(1, 0) + C_{mem}(\mu.\texttt{i}) + c_{call_2} \\ valid(\mu.\texttt{gas}, c_1, |\mathbf{s}| + 1) \qquad valid(\mu.\texttt{gas}, c_2, |\mathbf{s}| + 1)\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \rightarrow \begin{array}{l} EXC(\mu.\texttt{cond} \wedge \mathbf{va} = 0) :: (\mu, \iota, \sigma, \eta) :: S \\ EXC(\mu.\texttt{cond} \wedge 0 < \mathbf{va} \leq \sigma(\iota.\texttt{actor}).\texttt{b}) :: (\mu, \iota, \sigma, \eta) :: S \\ EXC(\mu.\texttt{cond} \wedge \mathbf{va} > \sigma(\iota.\texttt{actor}).\texttt{b})) :: (\mu, \iota, \sigma, \eta) :: S \end{array}}$$

$$\frac{(\omega_{\mu,\iota} = \texttt{CALL} \vee \omega_{\mu,\iota} = \texttt{CALLCODE}) \qquad |\mu.\mathbf{s}| < 7 \qquad \mathbf{cond} = \mu.\texttt{cond}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \rightarrow EXC(\mathbf{cond}) :: S}$$

If the execution of a `CALL` ends successfully, the output is written to $\mu.\texttt{rd}$. Notice that, even with a symbolic value, the gas cost is known because *va* satisfies some condition specified in the beginning of the call for which the gas cost is uniquely determined.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{CALL} \\ \mu.\mathbf{s} = g :: \mathbf{to} :: \mathbf{va} :: io :: is :: oo :: os :: \mathbf{s} \qquad \mathbf{to}_a = \mathbf{to} \mod 2^{160} \\ flag = (\sigma(\mathbf{to}_a) = \bot) \ ? \ 0 : 1 \qquad aw = M(M(\mu.\texttt{i}, io, is), oo, os) \\ c_{call} = C_{gascap}(\mathbf{va}, flag, g, \mu.\texttt{gas}) \qquad c = C_{base}(\mathbf{va}, flag) + C_{mem}(\mu.\texttt{i}) + c_{call} \\ \mu' = \mu[\texttt{gas} \ += g' - c][\texttt{pc} \ += 1][\texttt{m} \rightarrow \mu.\texttt{m}[[oo, oo + os - 1] \rightarrow \mathbf{d}]][\texttt{i} \rightarrow aw][\texttt{s} \rightarrow 1 :: \mathbf{s}][\texttt{rd} \rightarrow \mathbf{d}][\texttt{cond} \rightarrow \mathbf{cond}]\end{array}}{\Gamma \models HALT(\sigma', g', \mathbf{d}, \eta', \mathbf{cond}) :: (\mu, \iota, \sigma, \eta) :: S \rightarrow (\mu', \iota, \sigma', \eta') :: S}$$

When the execution of a call ends with `REVERT`, the remaining gas is refunded to the caller, the output is returned, the condition is updated and 0 is written in the stack of the caller to signal unsuccessful termination. The global state σ does not change.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{CALL} \\ \mu.\mathbf{s} = g :: \mathbf{to} :: \mathbf{va} :: io :: is :: oo :: os :: \mathbf{s} \qquad \mathbf{to}_a = \mathbf{to} \mod 2^{160} \\ flag = (\sigma(\mathbf{to}_a) = \bot) \ ? \ 0 : 1 \qquad aw = M(M(\mu.\texttt{i}, io, is), oo, os) \\ c_{call} = C_{gascap}(\mathbf{va}, flag, g, \mu.\texttt{gas}) \qquad c = C_{base}(\mathbf{va}, flag) + C_{mem}(\mu.\texttt{i}) + c_{call} \\ \mu' = \mu[\texttt{gas} \ += g' - c][\texttt{pc} \ += 1][\texttt{m} \rightarrow \mu.\texttt{m}[[oo, oo + os - 1] \rightarrow \mathbf{d}]][\texttt{i} \rightarrow aw][\texttt{s} \rightarrow 0 :: \mathbf{s}][\texttt{rd} \rightarrow \mathbf{d}][\texttt{cond} \rightarrow \mathbf{cond}]\end{array}}{\Gamma \models REV(g', \mathbf{d}, \mathbf{cond}) :: (\mu, \iota, \sigma, \eta) :: S \rightarrow (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{CALL} \\ \mu.\mathbf{s} = g :: \mathbf{to} :: \mathbf{va} :: io :: is :: oo :: os :: \mathbf{s} \qquad \mathbf{to}_a = \mathbf{to} \mod 2^{160} \\ flag = (\sigma(\mathbf{to}_a) = \bot) \ ? \ 0 : 1 \qquad aw = M(M(\mu.\texttt{i}, io, is), oo, os) \\ c_{call} = C_{gascap}(\mathbf{va}, flag, g, \mu.\texttt{gas}) \qquad c = C_{base}(\mathbf{va}, flag) + C_{mem}(\mu.\texttt{i}) + c_{call} \\ \mu' = \mu[\texttt{gas} \ -= c][\texttt{pc} \ += 1][\texttt{i} \rightarrow aw][\texttt{s} \rightarrow 0 :: \mathbf{s}][\texttt{rd} \rightarrow \epsilon][\texttt{cond} \rightarrow \mathbf{cond}]\end{array}}{\Gamma \models EXC(\mathbf{cond}) :: (\mu, \iota, \sigma, \eta) :: S \rightarrow (\mu', \iota, \sigma, \eta) :: S}$$

`CALLCODE` is very similar to `CALL`. This instruction also triggers the execution of a different contract, but the *actor* field of the execution environment is kept.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{CALLCODE} \qquad \mu.\mathbf{s} = g :: \mathbf{to} :: \mathbf{va} :: io :: is :: oo :: os :: \mathbf{s} \qquad |S| + 1 \leq 1024 \\ \mathbf{to}_a = \mathbf{to} \mod 2^{160} \qquad \sigma(\mathbf{to}_a) \neq \bot \qquad aw = M(M(\mu.\mathtt{i}, io, is), oo, os) \qquad \mathbf{d} = \mu.\mathtt{m}[io, io + is - 1] \\ c_{call_1} = C_{gascap}(0, 1, g, \mu.\mathtt{gas}) \qquad c_{call_2} = C_{gascap}(1, 1, g, \mu.\mathtt{gas}) \\ c_1 = C_{base}(0, 1) + C_{mem}(\mu.\mathtt{i}) + c_{call_1} \qquad c_2 = C_{base}(1, 1) + C_{mem}(\mu.\mathtt{i}) + c_{call_2} \\ valid(\mu.\mathtt{gas}, c_1, |\mathbf{s}| + 1) \qquad valid(\mu.\mathtt{gas}, c_2, |\mathbf{s}| + 1) \\ \mu'_1 = (c_{call_1}, 0, \lambda x.0, 0, \epsilon, \mu.\mathtt{rd}, \mu.\mathtt{cond} \wedge \mathbf{va} = 0) \\ \mu'_2 = (c_{call_2}, 0, \lambda x.0, 0, \epsilon, \mu.\mathtt{rd}, \mu.\mathtt{cond} \wedge 0 < \mathbf{va} \leq \sigma(\iota.\mathtt{actor}).\mathtt{b}) \\ \iota'_1 = \iota[\mathtt{input} \to \mathbf{d}][\mathtt{sender} \to \iota.\mathtt{actor}][\mathtt{value} \to 0][\mathtt{code} \to \sigma(to_a).\mathtt{code}] \\ \iota'_2 = \iota[\mathtt{input} \to \mathbf{d}][\mathtt{sender} \to \iota.\mathtt{actor}][\mathtt{value} \to \mathbf{va}][\mathtt{code} \to \sigma(to_a).\mathtt{code}] \end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to \begin{array}{l}(\mu'_1, \iota'_1, \sigma, \eta) :: (\mu, \iota, \sigma, \eta) :: S \\ (\mu'_2, \iota'_2, \sigma', \eta) :: (\mu, \iota, \sigma, \eta) :: S \\ EXC(\mu.\mathtt{cond} \wedge \mathbf{va} > \sigma(\iota.\mathtt{actor}).\mathtt{b})) :: (\mu, \iota, \sigma, \eta) :: S\end{array}}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{CALLCODE} \qquad \mu.\mathbf{s} = g :: \mathbf{to} :: \mathbf{va} :: io :: is :: oo :: os :: \mathbf{s} \qquad |S| + 1 \leq 1024 \\ \mathbf{to}_a = \mathbf{to} \mod 2^{160} \qquad \sigma(\mathbf{to}_a) = \bot \qquad aw = M(M(\mu.\mathtt{i}, io, is), oo, os) \qquad \mathbf{d} = \mu.\mathtt{m}[io, io + is - 1] \\ c_{call_1} = C_{gascap}(0, 1, g, \mu.\mathtt{gas}) \qquad c_{call_2} = C_{gascap}(1, 1, g, \mu.\mathtt{gas}) \\ c_1 = C_{base}(0, 1) + C_{mem}(\mu.\mathtt{i}) + c_{call_1} \qquad c_2 = C_{base}(1, 1) + C_{mem}(\mu.\mathtt{i}) + c_{call_2} \\ valid(\mu.\mathtt{gas}, c_1, |\mathbf{s}| + 1) \qquad valid(\mu.\mathtt{gas}, c_2, |\mathbf{s}| + 1) \\ \mu'_1 = (c_{call_1}, 0, \lambda x.0, 0, \epsilon, \mu.\mathtt{rd}, \mu.\mathtt{cond} \wedge \mathbf{va} = 0) \\ \mu'_2 = (c_{call_2}, 0, \lambda x.0, 0, \epsilon, \mu.\mathtt{rd}, \mu.\mathtt{cond} \wedge 0 < \mathbf{va} \leq \sigma(\iota.\mathtt{actor}).\mathtt{b}) \\ \iota'_1 = \iota[\mathtt{input} \to \mathbf{d}][\mathtt{sender} \to \iota.\mathtt{actor}][\mathtt{value} \to 0][\mathtt{code} \to \epsilon] \\ \iota'_2 = \iota[\mathtt{input} \to \mathbf{d}][\mathtt{sender} \to \iota.\mathtt{actor}][\mathtt{value} \to \mathbf{va}][\mathtt{code} \to \epsilon] \end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to \begin{array}{l}(\mu'_1, \iota'_1, \sigma, \eta) :: (\mu, \iota, \sigma, \eta) :: S \\ (\mu'_2, \iota'_2, \sigma', \eta) :: (\mu, \iota, \sigma, \eta) :: S \\ EXC(\mu.\mathtt{cond} \wedge \mathbf{va} > \sigma(\iota.\mathtt{actor}).\mathtt{b})) :: (\mu, \iota, \sigma, \eta) :: S\end{array}}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{CALLCODE} \qquad \mu.\mathbf{s} = g :: \mathbf{to} :: \mathbf{va} :: io :: is :: oo :: os :: \mathbf{s} \qquad |S| + 1 > 1024 \\ \mathbf{to}_a = \mathbf{to} \mod 2^{160} \qquad aw = M(M(\mu.\mathtt{i}, io, is), oo, os) \\ c_{call_1} = C_{gascap}(0, 1, g, \mu.\mathtt{gas}) \qquad c_{call_2} = C_{gascap}(1, 1, g, \mu.\mathtt{gas}) \\ c_1 = C_{base}(0, 1) + C_{mem}(\mu.\mathtt{i}) + c_{call_1} \qquad c_2 = C_{base}(1, 1) + C_{mem}(\mu.\mathtt{i}) + c_{call_2} \\ valid(\mu.\mathtt{gas}, c_1, |\mathbf{s}| + 1) \qquad valid(\mu.\mathtt{gas}, c_2, |\mathbf{s}| + 1) \end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to \begin{array}{l}EXC(\mu.\mathtt{cond} \wedge \mathbf{va} = 0) :: (\mu, \iota, \sigma, \eta) :: S \\ EXC(\mu.\mathtt{cond} \wedge 0 < \mathtt{va} \leq \sigma(\iota.\mathtt{actor}).\mathtt{b}) :: (\mu, \iota, \sigma, \eta) :: S \\ EXC(\mu.\mathtt{cond} \wedge \mathbf{va} > \sigma(\iota.\mathtt{actor}).\mathtt{b})) :: (\mu, \iota, \sigma, \eta) :: S\end{array}}$$

If there is not enough gas to send ether, but it is possible to execute a `CALLCODE` with value equal to 0, all possibilities are considered separately.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{CALLCODE} \qquad \mu.\mathbf{s} = g :: \mathbf{to} :: \mathbf{va} :: io :: is :: oo :: os :: \mathbf{s} \qquad |S| + 1 \leq 1024 \\ \mathbf{to}_a = \mathbf{to} \mod 2^{160} \qquad \sigma(\mathbf{to}_a) = \bot \qquad aw = M(M(\mu.\mathtt{i}, io, is), oo, os) \qquad \mathbf{d} = \mu.\mathtt{m}[io, io + is - 1] \\ c_{call_1} = C_{gascap}(0, 1, g, \mu.\mathtt{gas}) \qquad c_{call_2} = C_{gascap}(1, 1, g, \mu.\mathtt{gas}) \\ c_1 = C_{base}(0, 0) + C_{mem}(\mu.\mathtt{i}) + c_{call_1} \qquad c_2 = C_{base}(1, 0) + C_{mem}(\mu.\mathtt{i}) + c_{call_2} \\ valid(\mu.\mathtt{gas}, c_1, |\mathbf{s}| + 1) \qquad \neg valid(\mu.\mathtt{gas}, c_2, |\mathbf{s}| + 1) \\ \mu' = (c_{call_1}, 0, \lambda x.0, 0, \epsilon, \mu.\mathtt{rd}, \mu.\mathtt{cond} \wedge \mathbf{va} = 0) \\ \iota' = \iota[\mathtt{actor} \to to_a][\mathtt{input} \to \mathbf{d}][\mathtt{sender} \to \iota.\mathtt{actor}][\mathtt{value} \to 0][\mathtt{code} \to \epsilon] \end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to \begin{array}{l}(\mu', \iota', \sigma, \eta) :: (\mu, \iota, \sigma, \eta) :: S \\ EXC(\mu.\mathtt{cond} \wedge 0 < \mathbf{va} \leq \sigma(\iota.\mathtt{actor}).\mathtt{b})) :: S \\ EXC(\mu.\mathtt{cond} \wedge \mathbf{va} > \sigma(\iota.\mathtt{actor}).\mathtt{b})) :: (\mu, \iota, \sigma, \eta) :: S\end{array}}$$

If there is not enough gas to do any `CALLCODE`, three different exceptions are raised.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{CALLCODE} \qquad \mu.\mathbf{s} = g :: \mathbf{to} :: \mathbf{va} :: io :: is :: oo :: os :: \mathbf{s} \\ \mathbf{to}_a = \mathbf{to} \mod 2^{160} \qquad aw = M(M(\mu.\texttt{i}, io, is), oo, os) \\ c_{call} = C_{gascap}(0, 1, g, \mu.\texttt{gas}) \\ c = C_{base}(0,1) + C_{mem}(\mu.\texttt{i}) + c_{call} \qquad \neg valid(\mu.\texttt{gas}, c, |\mathbf{s}| + 1)\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \rightarrow \begin{array}{l} EXC(\mu.\texttt{cond} \wedge \mathbf{va} = 0) :: (\mu, \iota, \sigma, \eta) :: S \\ EXC(\mu.\texttt{cond} \wedge 0 < \mathbf{va} \leq \sigma(\iota.\texttt{actor}).\texttt{b})) :: S \\ EXC(\mu.\texttt{cond} \wedge \mathbf{va} > \sigma(\iota.\texttt{actor}).\texttt{b})) :: (\mu, \iota, \sigma, \eta) :: S \end{array}}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{CALLCODE} \\ \mu.\mathbf{s} = g :: \mathbf{to} :: \mathbf{va} :: io :: is :: oo :: os :: \mathbf{s} \qquad \mathbf{to}_a = \mathbf{to} \mod 2^{160} \\ aw = M(M(\mu.\texttt{i}, io, is), oo, os) \\ c_{call} = C_{gascap}(\mathbf{va}, 1, g, \mu.\texttt{gas}) \qquad c = C_{base}(\mathbf{va}, 1) + C_{mem}(\mu.\texttt{i}) + c_{call} \\ \mu' = \mu[\texttt{gas} \ += g' - c][\texttt{pc} \ += 1][\texttt{m} \rightarrow \mu.\texttt{m}[[oo, oo + os - 1] \rightarrow \mathbf{d}]][\texttt{i} \rightarrow aw][\texttt{s} \rightarrow 1 :: \mathbf{s}][\texttt{rd} \rightarrow \mathbf{d}][\texttt{cond} \rightarrow \mathbf{cond}]\end{array}}{\Gamma \models HALT(\sigma', g', \mathbf{d}, \eta', \mathbf{cond}) :: (\mu, \iota, \sigma, \eta) :: S \rightarrow (\mu', \iota, \sigma', \eta') :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{CALLCODE} \\ \mu.\mathbf{s} = g :: \mathbf{to} :: \mathbf{va} :: io :: is :: oo :: os :: \mathbf{s} \qquad \mathbf{to}_a = \mathbf{to} \mod 2^{160} \qquad aw = M(M(\mu.\texttt{i}, io, is), oo, os) \\ c_{call} = C_{gascap}(\mathbf{va}, 1, g, \mu.\texttt{gas}) \qquad c = C_{base}(\mathbf{va}, 1) + C_{mem}(\mu.\texttt{i} + c_{call}) \\ \mu' = \mu[\texttt{gas} \ += g' - c][\texttt{pc} \ += 1][\texttt{m} \rightarrow \mu.\texttt{m}[[oo, oo + os - 1] \rightarrow \mathbf{d}]][\texttt{i} \rightarrow aw][\texttt{s} \rightarrow 0 :: \mathbf{s}][\texttt{rd} \rightarrow \mathbf{d}][\texttt{cond} \rightarrow \mathbf{cond}]\end{array}}{\Gamma \models REV(g', \mathbf{d}, \mathbf{cond}) :: (\mu, \iota, \sigma, \eta) :: S \rightarrow (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{CALLCODE} \\ \mu.\mathbf{s} = g :: \mathbf{to} :: \mathbf{va} :: io :: is :: oo :: os :: \mathbf{s} \qquad \mathbf{to}_a = \mathbf{to} \mod 2^{160} \\ aw = M(M(\mu.\texttt{i}, io, is), oo, os) \\ c_{call} = C_{gascap}(\mathbf{va}, 1, g, \mu.\texttt{gas}) \qquad c = C_{base}(\mathbf{va}, 1) + C_{mem}(\mu.\texttt{i}) + c_{call} \\ \mu' = \mu[\texttt{gas} \ -= c][\texttt{pc} \ += 1][\texttt{i} \rightarrow aw][\texttt{s} \rightarrow 0 :: \mathbf{s}][\texttt{rd} \rightarrow \epsilon][\texttt{cond} \rightarrow \mathbf{cond}]\end{array}}{\Gamma \models EXC(\mathbf{cond}) :: (\mu, \iota, \sigma, \eta) :: S \rightarrow (\mu', \iota, \sigma, \eta) :: S}$$

`RETURN` has the same semantics: both arguments must be numeric and the memory contents can contain symbols.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{RETURN} \\ \mu.\mathbf{s} = io :: is :: \mathbf{s} \qquad aw = M(\mu.\texttt{i}, io, is) \qquad c = C_{mem}(\mu.\texttt{i}, aw) \\ valid(\mu.\texttt{gas}, c, |\mathbf{s}|) \qquad \mathbf{d} = \mu.\texttt{m}[io, io + is - 1] \qquad g = \mu.\texttt{gas} - c \qquad \mathbf{cond} = \mu.\texttt{cond}\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \rightarrow HALT(\sigma, g, \mathbf{d}, \eta, \mathbf{cond}) :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{RETURN} \\ \mu.\mathbf{s} = io :: is :: \mathbf{s} \qquad aw = M(\mu.\texttt{i}, io, is) \qquad c = C_{mem}(\mu.\texttt{i}, aw) \qquad \neg valid(\mu.\texttt{gas}, c, |s|)\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \rightarrow EXC(\mu.\texttt{cond}) :: S}$$

$$\frac{\omega_{\mu,\iota} = \texttt{RETURN} \qquad |\mu.\mathbf{s}| < 2}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \rightarrow EXC(\mu.\texttt{cond}) :: S}$$

Besides $\iota.\texttt{actor}$, it is also possible to keep the values of $\iota.\texttt{sender}$ and $\iota.\texttt{value}$ when calling another contract using `DELEGATECALL`. Since the `DELEGATECALL` operation does not receive a value in the parame-

ters, every argument is numeric – except possibly the address – and so the modification of the semantics is straightforward.

$$\frac{\begin{array}{cc}\omega_{\mu,\iota} = \texttt{DELEGATECALL} & \mu.\mathbf{s} = g :: \mathbf{to} :: io :: is :: oo :: os :: \mathbf{s} \\ \mathbf{to}_a = \mathbf{to} \mod 2^{160} \quad \sigma(\mathbf{to}_a) \neq \bot & aw = M(M(\mu.\mathtt{i}, io, is), oo, os) \quad \mathbf{d} = \mu.\mathtt{m}[io, io + is - 1] \\ c_{call} = C_{gascap}(0, 1, g, \mu.\mathtt{gas}) & c = C_{base}(0, 1) + C_{mem}(\mu.\mathtt{i}) + c_{call} \\ valid(\mu.\mathtt{gas}, c, |\mathbf{s}| + 1) & |S| + 1 \leq 1024 \\ \mu' = (c_{call}, 0, \lambda x.0, 0, \epsilon, \mu.\mathtt{rd}, \mu.\mathtt{cond}) & \iota' = \iota[\mathtt{input} \to \mathbf{d}][\mathtt{code} \to \sigma(to_a).\mathtt{code}] \end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota', \sigma, \eta) :: (\mu, \iota, \sigma, \eta) :: S}$$

$$\frac{\begin{array}{cc}\omega_{\mu,\iota} = \texttt{DELEGATECALL} & \mu.\mathbf{s} = g :: \mathbf{to} :: io :: is :: oo :: os :: \mathbf{s} \\ \mathbf{to}_a = \mathbf{to} \mod 2^{160} \quad \sigma(\mathbf{to}_a) = \bot & aw = M(M(\mu.\mathtt{i}, io, is), oo, os) \quad \mathbf{d} = \mu.\mathtt{m}[io, io + is - 1] \\ c_{call} = C_{gascap}(0, 1, g, \mu.\mathtt{gas}) & c = C_{base}(0, 1) + C_{mem}(\mu.\mathtt{i}) + c_{call} \\ valid(\mu.\mathtt{gas}, c, |\mathbf{s}| + 1) & |S| + 1 \leq 1024 \\ \mu' = (c_{call}, 0, \lambda x.0, 0, \epsilon, \mu.\mathtt{rd}, \mu.\mathtt{cond}) & \iota' = \iota[\mathtt{input} \to \mathbf{d}][\mathtt{code} \to \epsilon] \end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota', \sigma, \eta) :: (\mu, \iota, \sigma, \eta) :: S}$$

$$\frac{\begin{array}{cc}\omega_{\mu,\iota} = \texttt{DELEGATECALL} & \mu.\mathbf{s} = g :: \mathbf{to} :: io :: is :: oo :: os :: \mathbf{s} \\ \mathbf{to}_a = \mathbf{to} \mod 2^{160} & aw = M(M(\mu.\mathtt{i}, io, is), oo, os) \\ c_{call} = C_{gascap}(0, 1, g, \mu.\mathtt{gas}) & c = C_{base}(0, 1) + C_{mem}(\mu.\mathtt{i}) + c_{call} \\ valid(\mu.\mathtt{gas}, c, |\mathbf{s}| + 1) & |S| + 1 > 1024 \end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to EXC(\mu.\mathtt{cond}) :: (\mu, \iota, \sigma, \eta) :: S}$$

$$\frac{\begin{array}{cc}\omega_{\mu,\iota} = \texttt{DELEGATECALL} & \mu.\mathbf{s} = g :: \mathbf{to} :: io :: is :: oo :: os :: \mathbf{s} \\ \mathbf{to}_a = \mathbf{to} \mod 2^{160} & aw = M(M(\mu.\mathtt{i}, io, is), oo, os) \\ c_{call} = C_{gascap}(0, 1, g, \mu.\mathtt{gas}) & c = C_{base}(0, 1) + C_{mem}(\mu.\mathtt{i}) + c_{call} \\ \multicolumn{2}{c}{\neg valid(\mu.\mathtt{gas}, c, |\mathbf{s}| + 1)} \end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to EXC(\mu.\mathtt{cond}) :: S}$$

$$\frac{\omega_{\mu,\iota} = \texttt{DELEGATECALL} \quad |\mu.\mathbf{s}| < 6}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to EXC(\mu.\mathtt{cond}) :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{DELEGATECALL} \\ \mu.\mathbf{s} = g :: \mathbf{to} :: io :: is :: oo :: os :: \mathbf{s} \qquad \mathbf{to}_a = \mathbf{to} \mod 2^{160} \\ aw = M(M(\mu.\mathtt{i}, io, is), oo, os) \\ c_{call} = C_{gascap}(0, 1, g, \mu.\mathtt{gas}) \qquad c = C_{base}(0, 1) + C_{mem}(\mu.\mathtt{i}) + c_{call} \\ \mu' = \mu[\mathtt{gas} \ += g' - c][\mathtt{pc} \ += 1][\mathtt{m} \to \mu.\mathtt{m}[[oo, oo + os - 1] \to \mathbf{d}]][\mathtt{i} \to aw][\mathtt{s} \to 1 :: \mathbf{s}][\mathtt{rd} \to \mathbf{d}][\mathtt{cond} \to \mathbf{cond}] \end{array}}{\Gamma \models HALT(\sigma', g', \mathbf{d}, \eta', \mathbf{cond}) :: (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma', \eta') :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{DELEGATECALL} \\ \mu.\mathbf{s} = g :: \mathbf{to} :: io :: is :: oo :: os :: \mathbf{s} \qquad \mathbf{to}_a = \mathbf{to} \mod 2^{160} \qquad aw = M(M(\mu.\mathtt{i}, io, is), oo, os) \\ c_{call} = C_{gascap}(0, 1, g, \mu.\mathtt{gas}) \qquad c = C_{base}(0, 1) + C_{mem}(\mu.\mathtt{i} + c_{call}) \\ \mu' = \mu[\mathtt{gas} \ += g' - c][\mathtt{pc} \ += 1][\mathtt{m} \to \mu.\mathtt{m}[[oo, oo + os - 1] \to d]][\mathtt{rd} \to \mathbf{d}][\mathtt{i} \to aw][\mathtt{s} \to 0 :: \mathbf{s}][\mathtt{cond} \to \mathbf{cond}] \end{array}}{\Gamma \models REV(g', \mathbf{d}, \mathbf{cond}) :: (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{DELEGATECALL} \\ \mu.\mathbf{s} = g :: \mathbf{to} :: io :: is :: oo :: os :: \mathbf{s} \qquad \mathbf{to}_a = \mathbf{to} \mod 2^{160} \\ aw = M(M(\mu.\texttt{i}, io, is), oo, os) \\ c_{call} = C_{gascap}(0, 1, g, \mu.\texttt{gas}) \qquad c = C_{base}(0,1) + C_{mem}(\mu.\texttt{i}) + c_{call} \\ \mu' = \mu[\texttt{gas} \ -= c][\texttt{pc} \ += 1][\texttt{i} \to aw][\texttt{s} \to 0 :: \mathbf{s}][\texttt{rd} \to \epsilon][\texttt{cond} \to \mathbf{cond}]\end{array}}{\Gamma \models EXC(\mathbf{cond}) :: (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota, \sigma, \eta) :: S}$$

`SELFDESTRUCT` only receives one argument, an address, that must be numerical, so its semantics are analogous to the original ones, present in [6].

The `REVERT` instruction returns the remaining gas, output data and the condition to the caller.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{REVERT} \\ \mu.\mathbf{s} = io :: is :: \mathbf{s} \qquad aw = M(\mu.\texttt{i}, io, is) \qquad c = C_{mem}(\mu.\texttt{i}, aw) \\ valid(\mu.\texttt{gas}, c, |\mathbf{s}|) \qquad \mathbf{d} = \mu.\texttt{m}[io, io + is - 1] \qquad gas = \mu.\texttt{gas} - c \qquad \mathbf{cond} = \mu.\texttt{cond}\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to REV(g, \mathbf{d}, \mathbf{cond}) :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{REVERT} \\ \mu.\mathbf{s} = io :: is :: \mathbf{s} \qquad aw = M(\mu.\texttt{i}, io, is) \qquad c = C_{mem}(\mu.\texttt{i}, aw) \qquad \neg valid(\mu.\texttt{gas}, c, |\mathbf{s}|)\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to EXC(\mu.\texttt{cond}) :: S}$$

$$\frac{\omega_{\mu,\iota} = \texttt{REVERT} \qquad |\mu.\mathbf{s}| < 2}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to EXC(\mu.\texttt{cond}) :: S}$$