Providing formal semantics for Solidity to allow its verification

Dotación de una semántica formal a Solidity que permita su verificación

Pablo Páez Velasco

Trabajo de fin de grado
Grado en Ingeniería Informática

Universidad Complutense de Madrid

Facultad de Informática
Departamento de Sistemas Informáticos y Computación

Curso 2019 - 2020
Junio 2020

Directores:

Alberto Verdejo López
Narciso Martí Oliet

Resumen

En un contexto en el cual el número de lenguajes de programación cada vez es más elevado, y su uso se extiende a un mayor número de ámbitos, aparece la necesidad de asegurar la corrección o de determinar el resultado de ciertos programas. Debido a la falta de información concreta que muchas veces encontramos en la documentación oficial de los lenguajes, surgen técnicas como la verificación de programas a través de semánticas formales, que permiten modelizar mediante abstracciones matemáticas el funcionamiento de las instrucciones.

En este trabajo estudiaremos el lenguaje *Solidity*, utilizado para crear los llamados *smart contracts* en la plataforma *Ethereum*. Nuestro objetivo es estudiar qué propuestas de semánticas se han hecho, para posteriormente aportar la nuestra, que tratará de rellenar y corregir ciertos huecos e imprecisiones que hayamos podido encontrar en la literatura consultada.

En nuestro caso, presentamos una semántica centrada sobre todo en la obtención y actualización de valores en *storage*, una de las cuatro memorias que utiliza Ethereum.

Por último, el objetivo último de este trabajo es el ofrecer una semántica que sea implementable en lenguajes como *K* o *Maude*. En nuestro caso, hemos desarrollado diferentes módulos en Maude que pueden servir como base a la hora de realizar una verificación de un contrato en este lenguaje.

Palabras clave

Ethereum, Maude, Semántica operacional, Solidity, Storage

Abstract

Whilst the number of programming languages continuously increases, along with their multiple applications, the need of guaranteeing correction or predicting the outcome of a certain program arises. Due to the lack of information that is often available in the official documentation of the languages, formal methods seem a powerful tool in this sense. Thanks to program verification using formal semantics we can abstract the behaviour of an instruction using mathematical notation.

In this work we will study *Solidity*, a language used to create the so-called *smart contracts* in the *Ethereum* platform. Our goal is to study the state-of-the-art semantics in this topic, to ultimately suggest our own, with the aim of covering what we believe has not been covered yet, and to propose modifications to the rules that we have found rather imprecise.

We have decided to pay special attention to how values are updated and retrieved from the *storage*, which is one on the four data locations Solidity makes use of.

Lastly, it has been our goal to propose semantics that are executable in languages like *K* or *Maude*. For this reason, we have also coded several Maude modules which can be helpful when verifying a contract using this language.

Keywords

Ethereum, Maude, Operational Semantics, Solidity, Storage

Contents

1 Introduction **1**
1.1 Brief history of blockchain . . . 1
1.2 Our approach . . . 1
1.3 Literature . . . 3
1.4 Solidity memory model . . . 3
1.5 Syntax . . . 6
1.6 Example of a contract . . . 8
1.7 Notation . . . 9

2 Instruction rules **12**
2.1 Initial rules . . . 12
2.2 Function call rules . . . 15
2.2.1 Internal function calls . . . 15
2.2.2 External function calls . . . 17
2.3 Statements within the same function . . . 19
2.4 Declarations . . . 21
2.4.1 Declaring state variables . . . 21
2.4.2 Declaring local variables . . . 23
2.5 Assignments . . . 26
2.6 Events . . . 28
2.7 Arrays . . . 29

3 Storage **30**
3.1 Updating the storage . . . 30
3.2 Getting values from storage . . . 32

4 Evaluations **33**
4.1 Evaluating function calls . . . 33
4.1.1 Internal function calls . . . 33
4.1.2 External function calls . . . 35
4.2 Evaluating literals . . . 36
4.3 Evaluating array functions . . . 37
4.4 Evaluating expressions . . . 38
4.5 Address of an expression . . . 38

5 Auxiliary functions **39**
5.1 Typing expressions . . . 39
5.2 Distinguishing value-type and reference-type variables and expressions . . . 41
5.3 Distinguishing state and local variables . . . 41
5.4 Checking the location of a variable . . . 42
5.5 Offset . . . 43
5.6 Default value of a type . . . 44

5.7 Size 44

6 Example **45**

7 Implementation **55**
7.1 Value 55
7.2 Types 56
7.3 Global 57
7.4 EssentialFunctions 57
7.5 Aux_Ops 58
7.6 Storage 58
7.7 Memory 58
7.8 Evaluations and Transitions 58
7.9 Short Example 59

8 Conclusion **61**

References **63**

Agradecimientos

Muchas gracias a Alberto y a Narciso por su tiempo; por todas las correcciones, las horas y horas de tutorías y las decenas de correos respondiendo dudas sin las cuales este trabajo me hubiese sido imposible realizar.

Gracias a mis padres por cuidarme tanto y hacer posible así que me haya dedicado a los estudios de la forma en la que lo he hecho durante todos estos años. Gracias también por el cariño y por los consejos, sois el pilar sobre el que construyo todo.

A mis amigos y amigas, especialmente a Andy, Bittor, Irene, Julia y Pablo por su apoyo, por escuchar mis quejas y agobios, y por estar a mi lado aún cuando desaparezco a causa de los estudios. Gracias a mis amigas del Siglo por estar siempre.

A mis padres.

1 Introduction

1.1 Brief history of blockchain

Since 2008, due to the publication of the paper *Bitcoin: A Peer-to-Peer Electronic Cash*, by an unknown author, a new data structure has been gaining traction; the blockchain.

In broad terms, one can describe a blockchain as a data structure consisting of several blocks connected to one another creating a chain. The blocks are used to store information, creating what we call the *blockchain ledger*. In this structure, hashing techniques play an essential role.

The first implementation of blockchain occurred a year later, when *Bitcoin* was created. Since then, over 44,4 million users from all around the globe have joined Bitcoin [12].

Later on, many other applications started to emerge. Most of them either had a broader scope than Bitcoin, suggested a change in the protocol Bitcoin used, or both.

In this sense, in 2013 Vitalik Buterin proposed *Ethereum*, a blockchain platform in which users can create and run their own high-level scripts. These scripts are known as *smart contracts*, and they run on the *Ethereum Virtual Machine* (*EVM*).

Smart contracts are very similar to classes in object-oriented languages like Java. Each contract is assigned an address in the blockchain. Functions and variables can be declared within a contract. The variables declared in the contract are called *state variables*, and are accessible from every function in it. For each contract, a special type of function might be declared, the so-called *constructor* function, which is executed only after the contract is deployed to the blockchain.

In the same manner as Bitcoin had its own currency, used to reward its users for "mantaining the blockchain", Ethereum has its own currency, called Ether, which is obtained by the Ethereum users after running smart contracts.

There are many programming languages that can be used to create smart contracts. Some of them are *Mutan, LLL, Serpent, Viper, Lisk, Chain*, and *Solidity*. In this work, we will focus on the latter; our main goal is to review the existing literature focused on giving formal semantics for it, and suggest our own, based on the ideas we have seen.

1.2 Our approach

Before we dive in, there are two things that have to be taken into consideration. First of all, *Ethereum* translates the high-level code in smart contracts into *bytecode*; an intermediate language which resembles *machine code*. Since smart contracts are executed on the *EVM*, the protocol used by *Ethereum* suggests that functions should have a limited amount of "time" to perform, ensuring that every execution will finish —as we know, no algorithm can solve the halting problem—. To do so, functions receive money right before they start running, this money is called *gas*. Each operation consumes *gas*, and once a function spends all of its *gas*, it finishes.

In the *Ethereum Yellow Paper* [14] Gavin Wood stated the gas cost of every bytecode operation. As a consequence, almost all of the literature regarding semantics in *Ethereum* has focused on bytecode, resulting in a lack of papers about semantics for Solidity. We have decided to focus on high-level semantics because it is something that researchers have paid less attention to. We believe that it can be helpful for programmers, and it may be useful in order to get a deeper insight of the language, despite the fact that we will not be able to include *gas* in the equation. Nevertheless, the *gas* cost of an instruction could be estimated by translating it into bytecode.

The main issue when it comes to formalising Solidity is that it is a rather new language and therefore there is a lack of information about it. Some aspects are missing in the official documentation, and others are really hard to understand. Furthermore, it is not easy to run smart contracts, so it makes formalisation far more complicated. On top of all that, it is not a simple language; its syntax is very complex, comparable to those of Java or C++. Solidity deals with inheritance, function polymorphism, abstract contracts, interfaces, events, function and variables visibility... For this reason, we have decided to restrict the scope of this work to a subset of it.

In Section 2.3 we will be working with control sequences, such as *if-then-else* statements and loops. We will only consider the *while* loop as both the *do-while* and *for* can be seen as syntactic sugar for it. We will also consider internal and external function calls, both when used as statements (Section 2.2) or expressions to be evaluated (Section 4.1). Moreover, we will study variable declarations and assignments between them in Sections 2.4 and 2.5. In order to do this, we will need to consider the evaluation of expressions. We have needed a whole chapter in order to cover it fully (Chapter 4). Also, in Section 2.6 we will be covering events, which are key in order to communicate contracts, and something crucial when debugging, as they write information into the ledger, and no such thing as "writing on terminal" exists in Ethereum.

We will be omitting statements that deal directly with gas handling, as including gas into our semantics would be really challenging. Like we said before, in order to work with the gas cost of an instruction we would need to know the bytecode associated to every high-level instruction in Solidity. As we are not doing that, it is our belief that including any other operation dealing with gas, such as *assert*, *revert* or *send* would not be appropriate. If we were to include gas handling, the length of this work would increase considerably, along with the notation needed.

With the same purpose, we have decided to omit some global variables that can be used in Solidity, such as the amount of gas a certain user has, or the user who called a certain function, among many others. At the end of the day, these are just constants, and we would need to substantially expand the notation used, whilst not gaining much information about the language itself.

We will also be omitting auxiliary statements used to change the visibility of a function or a variable, its mutability, and some complex relationships between contracts, such as inheritance, or the use of abstract contracts and interfaces.

Another concept that Solidity includes which will not be addressed is *modifiers*; auxiliary functions that can be run before and after a certain function, used to assert that certain conditions are satisfied. We will not be studying them, as there is a fairly straightforward way to transform a contract that uses modifiers to one that does not, while in the other hand, it would be rather hard to give semantics for them.

1.3 Literature

After reading different papers which addressed the formalisation of Solidity ([1], [3], [5], [7], [8], [15], [16]), we have decided to pay special attention to three of them; "*Executable Operational Semantics of Solidity*" [7], "*VeriSolid: Correct-By-Design Smart Contracts for Ethereum*" [8] and "*SMT-Friendly Formalization of the Solidity Memory Model*" [5].

The first paper has been our main model; although we did not agree with some of the rules presented in it, we did like the way some key aspects were formalised. The second paper helped us a lot, as it extended the rules given by the first one. The third paper focused on giving a formalisation of the memory model, which has been a key piece of information, as it has been hard for us to find detailed information on this topic in the official documentation or other sources.

In this work, we have decided to pay special attention to the memory model, because there is a lack of high-level semantics about it for this language. It is important to state that there will be many things missing; it is completely beyond the scope of this work to give further details in a language like Solidity.

We have decided to use big-step semantics for this work; both [7] and [8] used them, so it seemed appropriate to maintain the same style. Moreover, it is our own preference to use them; we are interested in the resulting state of an execution, rather than each of the steps taken in order to achieve such state.

1.4 Solidity memory model

We will now see some basic concepts regarding the Solidity memory model. These concepts are key in order to understand the upcoming formalisation. Unlike other languages, Solidity uses four different data locations:

- **Storage**. Each contract has access to an array of 256-bit slots, which is stored in the blockchain. The variables stored in storage stay permanently.
- **Memory**. Here is where the contract stores all its local data used when executing functions. Its content is erased after the function finishes its execution.
- **Calldata**. Stores the incoming execution data provided to a function. It is non-modifiable.
- **Stack**. Used to load variables and store intermediate values generated by the EVM.

In this work, we will only focus on the first two. Studying the *Stack* would only make sense if we were to give semantics to Solidity's bytecode, and the *calldata* location plays a role that can be omitted. In fact, none of the papers we have consulted even mentioned it. We will study the storage in more detail in Section 3, focusing on how values are stored and retrieved from it.

When trying to formalise the memory, we encounter a problem; its behaviour is quite intricate, and there is a lack of specific information regarding its functioning. Given that situation, there are two possible ways to proceed:

- The first approach would be to assume its behaviour, by oversimplifying its model to an array or another simple structure, resulting in the memory and the storage being fairly similar. By doing this, one can give concrete semantics for the memory, although they will not be accurate.
- The second way would be not to assume anything about its behaviour, and keep the semantics more abstract. In any case, when implementing this model in a language like K or Maude, it would be possible to give more precise behaviour to the semantics, so that they are executable.

Most of the papers consulted opted for the first approach. In our case, we will use the latter, due to its duality between the abstract and concrete approach. It is our belief that this will make our semantics more versatile.

Lastly, there is some information regarding variable types and the way they are stored in storage that the reader has to be familiar with. We can distinguish two different variable types:

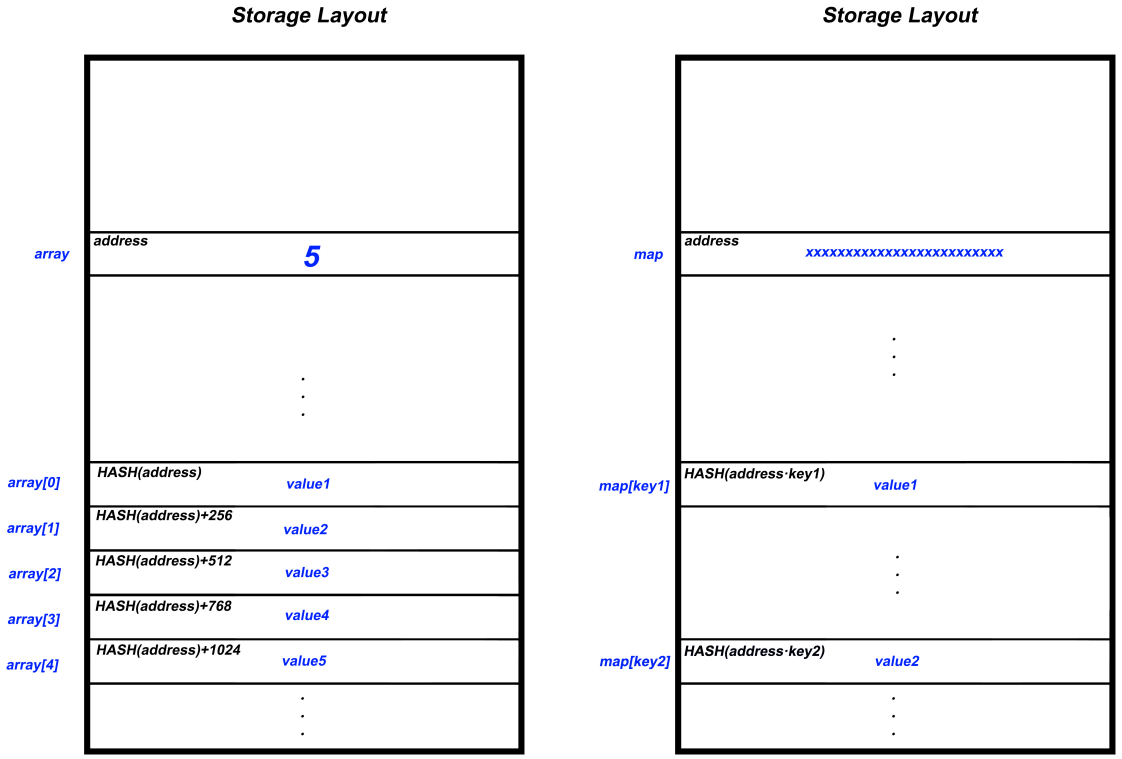
- **Simple types** or primitive types. We have only focused on some of the primitive types Solidity provides, which have been:
  - **bool**. They can only have a true or false value. It is the most simple type we will be working with.
  - **int8**, **int16**, ... **int256**. Signed integers, each one has a fixed size; from 8 to 256 bits.
  - **uint8**, **uint16**, ... **uint256**. Unsigned integers, like signed integers, each one has a fixed size; from 8 to 256 bits.
  - **int** and **uint**. Both are only aliases for int256 and uint256, respectively.
  - **address**. It is a 160-bit string, used to denote addresses in the blockchain ledger.
- **Complex types**. Unlike simple types, their structure is composite.
  - **Static Arrays**. They are given a fixed size, which cannot be changed. They are stored sequentially from their given address. All their elements must have the same type and can be accessed using [ ], like in many other modern languages.

– **Dynamic Arrays**. Unlike static arrays, they do not have a fixed size. In their given address, $addr_1$, Solidity stores their current size. Their elements will be stored sequentially starting from another address, which can be calculated by hashing $addr_1$.

– **Structs**. They consist of a number of members, each one can have their own type, and can be accessed in the same way as in other languages such as C or Java. They are stored sequentially from their given address.

– **Mappings**. Mappings are a very common type used in many languages, but in Solidity they act somehow different. Mappings do not store their keys, each element can be accessed by hashing the value resulting after concatenating the address associated with the mapping and the value of the key being accessed.

Both mappings and dynamic arrays make use of a hashing function in order to be stored. The algorithm used to hash values is *Keccak256*, although we will not study it in detail. In order to clarify, here we show two images to illustrate how both structures are lied out in the storage. Because we have not introduced any notation yet, this example is imprecise, but we believe that will help the reader to understand better.



Another concept that will appear is **literals**. Literals are constants, such as numbers or addresses, operations like $Exp_1 + Exp_2$, or expressions of the form $[Exp_1, Exp_2, \ldots, Exp_n]$ for array literals, $(Exp_1, Exp_2, \ldots, Exp_n)$ for struct literals or $[\{Exp_{c_1} : Exp_{v_1}\}, \{Exp_{c_2} : Exp_{v_2}\}, \ldots, \{Exp_{c_n} : Exp_{v_n}\}]$ for mapping literals. Unlike other expressions, which access elements of a variable, literals "define" an object. It is important to note that in the original Solidity syntax, struct literals are preceded by their type, like in other languages like C or Java. To make our semantics simpler, we have decided not to use that. We can assume there has been a preprocessing in the code so that the types are correct.

1.5 Syntax

In order to understand better the following chapters, it is useful to take a look at Solidity syntax. Our goal is not no give completely accurate semantics, but rather show the subset of Solidity that we have chosen to work with in a more formal way.

Each contract will be declared like so:

$$\text{contract } id\{Body\}$$

where *id* is the name of the contract. We will be using *id* to denote the identifier given to a variable, a function, and so on. *Body* contains all the definitions within the contract; thus, it can be described like so:

$$Body := Body.structs; Body.vars; Body.events; Body.functions.$$

Although in real contracts there might not be such a clear structure, the code could be rearranged so that it is divided into the sections presented above. Each of the four sections that are arranged sequentially, constituting the *Body* section, has its own structure.

Body.structs contains all the struct declarations. Inside each struct, there will be several member declarations, given by a type and the name of the member.

$$\begin{aligned} Body.structs := \ & \text{struct } id\{\text{T } id;\ \text{T } id;\ \ldots;\ \text{T } id;\} \\ & \text{struct } id\{\text{T } id;\ \text{T } id;\ \ldots;\ \text{T } id;\} \\ & \vdots \\ & \text{struct } id\{\text{T } id;\ \text{T } id;\ \ldots;\ \text{T } id\}. \end{aligned}$$

Following the struct declaration block, *Body.vars* declares all the contract's state variables. The parts appearing inside the brackets may be omitted, meaning that the variable is not being initialised.

$$Body.vars := \text{T } id[= \text{Exp}];\ \text{T } id[= \text{Exp}];\ \ldots;\ \text{T } id[= \text{Exp}];.$$

After the struct and variable declarations, the events will be declared. Events are used to log information into the ledger. We will discuss them in more detail in Section 2.6.

$$\begin{aligned} \textit{Body.events} := \text{event } \textit{id}(\text{T } \textit{id},\ \text{T } \textit{id},\ \ldots,\ \text{T } \textit{id});\\ \text{event } \textit{id}(\text{T } \textit{id},\ \text{T } \textit{id},\ \ldots,\ \text{T } \textit{id});\\ \vdots\\ \text{event } \textit{id}(\text{T } \textit{id},\ \text{T } \textit{id},\ \ldots,\ \text{T } \textit{id});. \end{aligned}$$

Lastly, the function definitions are stated. Note that, apart from the functions that the user may define, there will be two more functions: the constructor, declared with the word "constructor", and the *fallback function*, an auxiliary function that will be executed in case a non-existing function is called (see Section 2.2.1).

$$\begin{aligned} \textit{Body.functions} := [\text{constructor } (\text{T } \textit{id},\ \text{T } \textit{id},\ \ldots,\ \text{T } \textit{id})\{\ \text{Stmt}\}]\\ [\text{function } ()\{\ \text{Stmt}\}]\\ \text{function } \textit{id}(\text{T } \textit{id},\ \text{T } \textit{id},\ \ldots,\ \text{T } \textit{id})\{\ \text{Stmt}\}\\ \text{function } \textit{id}(\text{T } \textit{id},\ \text{T } \textit{id},\ \ldots,\ \text{T } \textit{id})\{\ \text{Stmt}\}\\ \vdots\\ \text{function } \textit{id}(\text{T } \textit{id},\ \text{T } \textit{id},\ \ldots,\ \text{T } \textit{id})\{\ \text{Stmt}\}. \end{aligned}$$

The syntax regarding types can be easily inferred from Section 1.4. In this syntax, we use T_2 for simple types and T for complex types.

$$\begin{aligned} &\text{T} := \text{T}_2 \mid \text{mapping}(\text{T}_2 \Rightarrow \text{T}) \mid \text{T}[] \mid \text{T}[n] \mid S\\ &\text{T}_2 := \text{bool} \mid \text{int8} \mid \text{int16} \mid \ldots \mid \text{int256} \mid \text{uint8} \mid \text{uint16} \mid \ldots \mid \text{uint256} \mid \text{uint} \mid \text{int} \mid \text{address}. \end{aligned}$$

This is the most important part in our syntax, as it gives precise information about the instructions that will be handled throughout the text.

$$\begin{aligned} \text{Stmt} := {} & \text{Stmt; Stmt} \mid T\ [M \mid S]\ \textit{id}[= \text{Exp}] \mid \text{Exp} = \text{Exp} \mid \text{while(Exp)\{Stmt\}} \mid \text{if(Exp)\{Stmt\}} \mid\\ & \text{if(Exp)\{Stmt\}else\{Stmt\}} \mid \text{return} \mid \text{return Exp} \mid \text{Exp.push()} \mid \text{Exp.push(Exp)} \mid \text{Exp.pop()} \mid\\ & \text{emit } \textit{id}(\text{Exp},\ \ldots,\ \text{Exp}) \mid \text{call } \textit{id}(\text{Exp},\ \ldots,\ \text{Exp}) \mid\\ & \text{call } \textit{id.id}.\text{value(Exp).gas(Exp)(Exp},\ \ldots,\ \text{Exp}). \end{aligned}$$

The expression $T\ \{M \mid S\}\ \textit{id}[= \text{Exp}]$ symbolises the declaration of a variable. As we will see in Section 2.4.2, an M or an S is added to denote whether the variable is stored in the memory or the storage. Moreover, variables can be given a certain value, from the evaluation of an expression.

The statements *push* and *pop* are used to insert or delete an element in a dynamic array, respectively, as we will see in Section 2.7.

The statement *emit* is used to call events, which are used to deploy information to the ledger. We will study them in Section 2.6.

The last two statements are used to call internal and external functions, respectively. We will cover those in Section 2.2. For now, it is enough to know that when calling a function, we will be using the auxiliary statement *call*. This is not included in the original semantics of Solidity. But we have decided to add it in our semantics as we believe it helps us differentiate between function calls used as expressions or statements.

$$\begin{aligned}\text{Exp} := {} & id \mid cte \mid \text{Exp } op \text{ Exp} \mid \text{Exp[Exp]} \mid \text{Exp}.id \mid [\text{Exp}, \ldots, \text{Exp}] \mid (\text{Exp}, \ldots, \text{Exp}) \mid \\ & [\{\text{Exp} : \text{Exp}\}, \ldots, \{\text{Exp} : \text{Exp}\}] \mid \text{Exp.size()} \mid \text{e-call } id(\text{Exp}, \ldots, \text{Exp}) \mid \\ & \text{e-call } id.id.\text{value(Exp).gas(Exp)(Exp}, \ldots, \text{Exp}).\end{aligned}$$

In the expressions syntax we have used *cte* to symbolise a constant. Exp *op* Exp is used to represent operations. The expression Exp.size() is used to retrieve the size of a given array. Finally, both function calls are used to obtain a certain value, returned by the function. Note that in this case, the function call is preceded by *e-call* instead of *call*.

1.6 Example of a contract

After giving a quick formal approximation to Solidity, in this section we will be looking at an example, with a view to clarifying the reader's doubts. We will be using this example in Section 6 to show how some of the rules presented in this work are handled.

```
contract company {

    struct worker {
        uint salary;
        bool working;
        uint8 area;
        address identifier;
    }

    struct shift {
        int16 number_hours;
    }

    uint company_number = 1;
    address boss_identifier;
    mapping(address => worker) workers;
    shift[] shifts;

    event salary_updated(address worker, uint new_salary);
    event is_factory_open(bool isopen);

    constructor (address a){
        boss_identifier = a;
```

```
        worker S boss = worker(1, false, 5, boss_identifier);
        workers[boss_identifier] = boss;
        shifts.push(shift(5));
        bool M b  = comp_areas(boss_identifier, 1);
    }

    function comp_areas(address ident, uint8 area_num) returns (bool){
         return area_num > workers[ident].area;
    }
}
```

Example of contract code

The code shows a contract used to symbolise an imaginary company. This contract uses two structs:

- *worker*, used to represent each person working for the company, with their salary, whether they are currently working or not, the area in which they work at, and their identifier.
- *shift*, used to represent a working shift.

The company has four state variables; the company number, the boss identifier, a mapping, which links every worker's identifier to their information, and an array which keeps all the working shifts.

There are two different events: one is used to notify that a salary has been updated, and another one to notify whether the factory is open or not.

The constructor sets the boss identifier, and adds him into the mapping. Adds his working shift, and checks if the area number he has been assigned to is smaller than 1.

The function *comp_areas* receives a worker's identifier and a certain area number, and returns whether the area number the worker has been assigned to is smaller than that number.

Note that before the code section for function *comp_areas*, the returning value is stated. We have omitted that information in our syntax, and we will be omitting it in our semantics for simplicity. As we have said before, we can assume that the code has been preprocessed in order to check consistency.

1.7 Notation

In this section we will be giving some details about the main notation that we will be using. Keep in mind that this is not everything that is needed for the formalisation, and that further details will be presented when needed. To distinguish between the rules that we propose and the rest of them, we have decided to use a color code:

- The rules and functions written in red are new rules, that do not appear in the papers consulted, although the structure of some of them might have been

influenced by the literature read.

- The rules and functions appearing in blue are either a modified version of some rule that appears in the articles read, or its structure is so similar to the structure used in other papers that we think it would be unfair to say it is ours completely.
- Lastly, we will use green when presenting a rule that is completely or almost completely like some rule in the papers consulted, or when giving rules appearing in semantics textbooks for simple, imperative languages.

From Section 5.2 onwards, all the rules are new, but we have decided not to use the color red for them, as it is easier to read.

We will be using σ to denote a smart contract instance. This instance is in fact a tuple $\sigma = (\Psi, M, \Omega)$.

Its first two elements are functions: $\Psi : \mathbb{P} \to \mathbb{B}$ denotes its storage configuration, while $M : \mathbb{P} \to \mathbb{B}$ denotes its memory configuration. Both map positive numbers (addresses), represented by $\mathbb{P}$, to bytes (the information contained in such addresses), represented by $\mathbb{B}$. Its last element Ω, is a *stack* of storage configurations to handle external function calls.

Both Ψ and M have their own name and type spaces. Therefore, we define two auxiliary functions for each one:

$$N_\Psi : \mathbb{ID}_\Psi \to \mathbb{P} \qquad N_M : \mathbb{ID}_M \to \mathbb{P},$$

mapping variable names to their addresses; and

$$\tau_\Psi : \mathbb{ID}_\Psi \to \mathbb{T} \qquad \tau_M : \mathbb{ID}_M \to \mathbb{T},$$

mapping variable names to their types.

We will be representing the sets containing the names of the variables defined as follows:

- $\mathbb{ID}_\Psi$. To represent the set containing the ids of the variables defined in storage.
- $\mathbb{ID}_M$. Containing the ids of the variables defined in memory.
- $\mathbb{ID} = \mathbb{ID}_\Psi \cup \mathbb{ID}_M$.

Another set that we will use is $\mathbb{ID}_\emptyset$, containing the names of the variables which are not declared in storage or in memory, like struct members or function arguments.

We will denote an execution step as $\langle(\sigma, x), Stmt\rangle \to \langle(\sigma', x'), v\rangle$, where x and x' are the execution status before and after executing the statement *Stmt*. There are four possible status:

- N, meaning normal execution.
- $R[v]$ meaning that we have encountered a *return* statement and the value v is

being returned.

- $R[\cdot]$ meaning that we have encountered a *return* statement, not followed by any value.
- E, meaning that an exception has occurred.

After executing the statement, we obtain a value denoted by v.

$\sigma_{functions}$ denotes the set of the names of the functions declared within the contract. Similarly, σ_{events} denotes the set of the names of the events declared within the contract.

We also need to map function names to their respective code and each argument's type they take. Therefore we define two functions:

$$\Gamma_p : \sigma_{functions} \to \mathbb{C} \qquad \Gamma_t : \sigma_{functions} \to (\mathbb{ID}_\emptyset, \mathbb{T})^n$$

Γ_p maps each function's name to its code, and Γ_t maps each name to a tuple of types, one for each argument the function takes, in the same order as defined in the code. $\mathbb{C}$ denotes the set containing the code for each function. Finally, $\mathbb{T}$ is the set containing every type declared in the contract.

Some other important sets we need to define are:

- $\mathbb{S}$, the set containing every struct type defined in our contract.
- $\mathcal{SV}$, containing the name of every state variable defined.

Before introducing notation regarding addresses, we will show four more auxiliary functions.

- $\mathcal{V} : \mathbb{ID} \to \{static, dynamic\}$, a partial function mapping vector variable names to their type, either static or dynamic.
- $\mathcal{S} : \mathbb{S} \to (\mathbb{ID}_\emptyset \times \mathbb{T})^n$, mapping struct names to tuples of pairs, one for each "component" of the struct. Thus, for each struct type we would get a tuple

$$\langle (id_1, \mathrm{T}_1), (id_2, \mathrm{T}_2), \ldots, (id_n, \mathrm{T}_n) \rangle.$$

- $\mathcal{M} : \mathbb{ID} \to \mathbb{ID}^n$, mapping the name of a map to every key stored in it.
- Trad(), which translates syntactic symbols to semantic symbols. In this sense,

$$\mathrm{Trad}(||) = \vee,\ \mathrm{Trad}(\&\&) = \wedge,\ \mathrm{Trad}(\mathtt{1}) = 1,\ \mathrm{Trad}(\mathtt{+}) = +.$$

The last thing that needs to be shown before we start showing the semantic rules is the notation used when using addresses. Keep in mind that the following functions will only be used when working with storage, not memory. We will use three different functions:

- $\lceil \text{addr} \rceil^{(l, Type)}$ will be used to return the first address that can be used to store a variable of type *Type*, provided that the first available address is addr, and the block size is l. This is important, as variables need to be aligned (Section 3).

$$\lceil \text{addr} \rceil^{(l, PrimType)} = \begin{cases} \text{addr} & (\text{addr} \, mod \, l) + \text{Size}\,(PrimType) \leq l \\ (\lfloor \text{addr}/l \rfloor + 1) \cdot l & \text{otherwise} \end{cases},$$

$$\lceil \text{addr} \rceil^{(l, Type)} = \begin{cases} \text{addr} & \text{addr} \, mod \, l = 0 \\ (\lfloor \text{addr}/l \rfloor + 1) \cdot l & \text{otherwise} \end{cases}.$$

- $\text{addr} \uparrow (l, Type)$. This function will be used to update the pointer Λ, which will contain the first free address in storage. The returned value will be the first free address after saving an element of type *Type*, given that the block size is l.

$$\text{addr} \uparrow (l, Type) = \lceil \text{addr} \rceil^{(l, Type)} + \text{Size}\,(Type).$$

- $[addr]_{S_\sigma}^{Size}$ will return the value stored in the address *addr* to *addr* + *Size* − 1. It will be used to retrieve the values stored in storage.

In Solidity, the current block size is 32 bytes. So we could assume that $l = 32 \cdot 8 = 256$.

2 Instruction rules

After the introduction to Solidity and our approach, we will begin with the actual rule definitions. We have decided to omit numerous rules for handling errors. This is due to the fact that including them would increase the length of this text considerably, and it would make it far more difficult to read and understand. It is our belief that the reader will easily see where the rules are missing, and how they should be written, based on the rules that handle errors presented in this work.

2.1 Initial rules

The first thing we need is a rule that will trigger the verification tool; a starting point from where the rest of the rules will be invoked. In our work, there will be an initial rule, used to deploy a new contract to the blockchain. We are starting the execution from "outside", as an Ethereum transaction, rather than from Solidity code itself.

As it appears in Solidity's documentation, contracts can be created both "from outside", as an Ethereum transaction, or from within a Solidity contract. Due to the lack of information provided in the official documentation regarding the latter way, we will only give rules for the first approach. In any case, we believe that this approach is easily extendable, so that it can be modified in order to fit both ways.

We will begin by transforming the syntax used in Section 1.5 into rigorous notation. As we showed earlier, contracts can be created by writing:

$$\text{contract } c\{Body\}.$$

Inside the *Body* section, there are four different parts:

$$Body := Body.structs; Body.vars; Body.events; Body.functions.$$

Where each part has been already described in Section 1.5.

In this case, we will be giving more formal notation for each of the types and identifiers, in order to give more details about the auxiliary functions and sets described in Section 1.7. To distinguish between the types given for structs, event arguments, and function arguments, we will be using ${}_s\mathrm{T}$, ${}_e\mathrm{T}$ and ${}_f\mathrm{T}$ respectively. The same approach will be used with the notation used for identifiers. We are aware that this notation does indeed make the notation more complicated, but we believe that it is key to give rigorous notation for this.

$$\begin{aligned} Body.structs := \ & \text{struct } S_1\{{}_s\mathrm{T}_1^1\ {}_sid_1^1;\ {}_s\mathrm{T}_2^1\ {}_sid_2^1;\ \ldots;\ {}_s\mathrm{T}_{n_1}^1\ {}_sid_{n_1}^1;\} \\ & \text{struct } S_2\{{}_s\mathrm{T}_1^2\ {}_sid_1^2;\ {}_s\mathrm{T}_2^2\ {}_sid_2^2;\ \ldots;\ {}_s\mathrm{T}_{n_2}^2\ {}_sid_{n_2}^2;\} \\ & \quad \vdots \\ & \text{struct } S_k\{{}_s\mathrm{T}_1^k\ {}_sid_1^k;\ {}_s\mathrm{T}_2^k\ {}_sid_2^k;\ \ldots;\ {}_s\mathrm{T}_{n_k}^k\ {}_sid_{n_k}^k;\}. \end{aligned}$$

For the types and identifiers used in the state variable declarations, we do not have to use any letter sub-index.

$$Body.vars := \mathrm{T}_1\ id_1[= Exp_1];\ \mathrm{T}_2\ id_2[= Exp_2];\ \ldots;\ \mathrm{T}_n\ id_n[= Exp_n];.$$

Note that we have replaced *id* by S_i for struct identifiers, e_i for event identifiers, and f_i for function identifiers, with the aim to distinguish them.

$$\begin{aligned} Body.events := \ & \text{event } e_1({}_e\mathrm{T}_1^1\ {}_eid_1^1,\ {}_e\mathrm{T}_2^1\ {}_eid_2^1,\ \ldots,\ {}_e\mathrm{T}_{n_1}^1\ {}_eid_{n_1}^1); \\ & \text{event } e_2({}_e\mathrm{T}_1^2\ {}_eid_1^2,\ {}_e\mathrm{T}_2^2\ {}_eid_2^2,\ \ldots,\ {}_e\mathrm{T}_{n_2}^2\ {}_eid_{n_2}^2); \\ & \quad \vdots \\ & \text{event } e_{k'}({}_e\mathrm{T}_1^{k'}\ {}_eid_1^{k'},\ {}_e\mathrm{T}_2^{k'}\ {}_eid_2^{k'},\ \ldots,\ {}_e\mathrm{T}_{n_{k'}}^{k'}\ {}_eid_{n_{k'}}^{k'});. \end{aligned}$$

Lastly, this is the notation used for each of the functions:

$$\begin{aligned} \mathit{Body.functions} := [&\text{constructor } ({}_f\mathrm{T}_1^c\ {}_f id_1^c,\ {}_f\mathrm{T}_2^c\ {}_f id_2^c,\ \ldots,\ {}_f\mathrm{T}_{n_c}^c\ {}_f id_{n_c}^c)\{\mathit{Block}_c\}] \\ &[\text{function } ()\{\mathit{Block}_{fallback}\}] \\ &\text{function } f_1({}_f\mathrm{T}_1^1\ {}_f id_1^1,\ {}_f\mathrm{T}_2^1\ {}_f id_2^1,\ \ldots,\ {}_f\mathrm{T}_{n_1}^1\ {}_f id_{n_1}^1)\{\mathit{Block}_1\} \\ &\text{function } f_2({}_f\mathrm{T}_1^2\ {}_f id_1^2,\ {}_f\mathrm{T}_2^2\ {}_f id_2^2,\ \ldots,\ {}_f\mathrm{T}_{n_2}^2\ {}_f id_{n_2}^2)\{\mathit{Block}_2\} \\ &\qquad\vdots \\ &\text{function } f_{k''}({}_f\mathrm{T}_1^{k''}\ {}_f id_1^{k''},\ {}_f\mathrm{T}_2^{k''}\ {}_f id_2^{k''},\ \ldots,\ {}_f\mathrm{T}_{n_{k''}}^{k''}\ {}_f id_{n_{k''}}^{k''})\{\mathit{Block}_{k''}\}. \end{aligned}$$

If there is no constructor in the code provided to declare the contract, the system will assume the default constructor, which is constructor(). The same thing would happen if no fallback function is declared. Although the code inside the constructor does not include a *return* statement, for simplicity we will assume there is one. This will allow us to treat the constructor as if it were a normal function.

We are aware that the notation used can be rather complex at first, but we will not be going over it again in the text, and it is certainly useful in order to avoid any abuse of notation.

$$\textbf{INIT}\ \frac{\begin{array}{c} \mathit{initialise}(\mathit{Body}) = \sigma \\ \langle(\sigma, N), \mathit{Body.vars}\rangle \to \langle(\sigma', N), \cdot\rangle \end{array}}{\langle \text{contract } c\{\mathit{Body}\}, (v_1, v_2, \ldots, v_{n_c})\rangle \Rightarrow \langle(\sigma', N), c(v_1, v_2, \ldots, v_{n_c})\rangle}$$

As a new contract is being created, it will be assigned a new address in the blockchain. Once that is done, we will create a new state instance, σ, from where the execution will start. Before processing all the definitions stated in the contract, we will initialise the auxiliary sets and functions. To finish, the constructor will be executed.

The new state will be initialised based on the contract's definition. Thus, its sets and functions will be initialised as follows:

$$\begin{aligned} \mathbb{ID}_\Psi &= \{id_1, id_2, \ldots, id_n\}, \\ \mathbb{ID}_M &= \emptyset, \\ \sigma_{functions} &= \{c, f_1, f_2, \ldots, f_{k''}\}, \\ \sigma_{events} &= \{e_1, e_2, \ldots, e_{k'}\}, \\ \mathbb{S} &= \{S_1, S_2, \ldots, S_k\}, \\ \mathcal{SV} &= \{id_1, id_2, \ldots, id_n\}. \end{aligned}$$

Please note that, although $\mathcal{SV}$ and $\mathbb{ID}_\Psi$ are the same at the moment, $\mathcal{SV}$ will remain unchanged, as it contains only the name of every state variable, while $\mathbb{ID}_\Psi$ contains the name of the variables stored in storage.

The auxiliary functions defined in Section 1.7 will have the following behaviour:

$$\begin{aligned}
\Gamma_p(c) &= \mathit{Block}_c, \\
\Gamma_p(f_i) &= \mathit{Block}_i, \quad 1 \le i \le k'', \\
\Gamma_t(c) &= ((_f id_1^c, {}_f\mathrm{T}_1^c), (_f id_2^c, {}_f\mathrm{T}_2^c), \ldots, (_f id_{n_c}^c, {}_f\mathrm{T}_{n_c}^c)), \\
\Gamma_t(f_i) &= ((_f id_1^i, {}_f\mathrm{T}_1^i), (_f id_2^i, {}_f\mathrm{T}_2^i), \ldots, (_f id_{n_i}^i, {}_f\mathrm{T}_{n_i}^i)), \quad 1 \le i \le k'', \\
\mathcal{S}(S_i) &= ((_s id_1^i, {}_s\mathrm{T}_1^i), (_s id_2^i, {}_s\mathrm{T}_2^i), \ldots, (_s id_{n_i}^i, {}_s\mathrm{T}_{n_i}^i)), \quad 1 \le i \le k.
\end{aligned}$$

Finally, the initial value of Λ, the pointer containing the address of the first free slot in the storage, will be 0.

2.2 Function call rules

Before executing a function, each argument given to it is saved in the memory. After that, the code of the function is run.

Function calls can be external or internal. Internal function calls are calls to functions within the same contract, while external function calls trigger the execution of a function defined in another contract.

2.2.1 Internal function calls

When giving semantics to internal function calls, we have decided to separate them into two different groups: while FUN-CALL and FUN-CALL-EXC take

$$\text{“call } \mathit{name}(\mathit{Exp}_1, \mathit{Exp}_2, \ldots, \mathit{Exp}_n)\text{”}$$

statements, the others only take the function name and its arguments. The reason to do this is to simplify the rules as much as possible.

When calling a function, we first evaluate each expression given as an argument, and replace them with their respective values.

$$\textbf{FUN-CALL}\ \frac{\begin{array}{c}\mathrm{Eval}(\sigma, \mathit{Exp}_1) \to \langle(\sigma_1, N), v_1\rangle \\ \mathrm{Eval}(\sigma_1, \mathit{Exp}_2) \to \langle(\sigma_2, N), v_2\rangle \\ \vdots \\ \mathrm{Eval}(\sigma_{n-1}, \mathit{Exp}_n) \to \langle(\sigma_n, N), v_n\rangle \\ \langle(\sigma_n, N), \mathit{name}(v_1, v_2, \ldots, v_n)\rangle \to \langle(\sigma', N), \cdot\rangle\end{array}}{\langle(\sigma, N), \text{call } \mathit{name}(\mathit{Exp}_1, \mathit{Exp}_2, \ldots, \mathit{Exp}_n)\rangle \to \langle(\sigma', N), \cdot\rangle}$$

$$\textbf{FUN-CALL-EXC}_1\ \frac{\begin{array}{c}\mathrm{Eval}(\sigma, \mathit{Exp}_1) \to \langle(\sigma_1, N), v_1\rangle \\ \mathrm{Eval}(\sigma_1, \mathit{Exp}_2) \to \langle(\sigma_2, N), v_2\rangle \\ \vdots \\ \mathrm{Eval}(\sigma_{n-1}, \mathit{Exp}_n) \to \langle(\sigma_n, N), v_n\rangle \\ \langle(\sigma_n, N), \mathit{name}(v_1, v_2, \ldots, v_n)\rangle \to \langle(\sigma', E), \cdot\rangle\end{array}}{\langle(\sigma, N), \text{call } \mathit{name}(\mathit{Exp}_1, \mathit{Exp}_2, \ldots, \mathit{Exp}_n)\rangle \to \langle(\sigma', E), \cdot\rangle}$$

$$\textbf{FUN-CALL-EXC}_2 \quad \frac{\begin{array}{c}\text{Eval}(\sigma, Exp_1) \to \langle(\sigma_1, x_1), v_1\rangle \\ \text{Eval}(\sigma_1, Exp_2) \to \langle(\sigma_2, x_2), v_2\rangle \\ \vdots \\ \text{Eval}(\sigma_{n-1}, Exp_n) \to \langle(\sigma_n, x_n), v_n\rangle \\ x_1 = E \ \vee x_2 = E \ \vee \cdots \vee x_n = E\end{array}}{\langle(\sigma, N), \text{call } name(Exp_1, Exp_2, \ldots, Exp_n)\rangle \to \langle(\sigma_n, E), \cdot\rangle\rangle}$$

After the evaluation of the arguments, the function code is executed. To do so, we first check the function is defined within the contract, then we get its code and the type of each argument the function takes, using Γ_p and Γ_t. Finally, the code is executed after declaring the arguments in the memory.

Here we do not distinguish between functions that return values and those which do not return anything. This is because these rules are applied when executing statements, not when evaluating expressions. We will study those cases later on, in Section 4.1. If no exception is raised, the program will continue its execution normally.

$$\textbf{TRANSITION} \quad \frac{\begin{array}{c}name \in \sigma_{functions}, \\ \Gamma_p\, name = Block, \qquad \Gamma_t\, name = ((p_1, \mathrm{T}_1), (p_2, \mathrm{T}_2), \ldots, (p_n, \mathrm{T}_n)) \\ \langle(\sigma, N), \mathrm{T}_1\,\mathrm{M}\,p_1 = v_1;\ \mathrm{T}_2\,\mathrm{M}\,p_2 = v_2; \ldots;\ \mathrm{T}_n\,\mathrm{M}\,p_n = v_n; Block\rangle \to \langle(\sigma', R[\cdot]), \cdot\rangle\end{array}}{\langle(\sigma, N), name(v_1, v_2, \ldots, v_n)\rangle \to \langle(\sigma', N), \cdot\rangle}$$

$$\textbf{TRANSITION-RET} \quad \frac{\begin{array}{c}name \in \sigma_{functions}, \\ \Gamma_p\, name = Block, \qquad \Gamma_t\, name = ((p_1, \mathrm{T}_1), (p_2, \mathrm{T}_2), \ldots, (p_n, \mathrm{T}_n)) \\ \langle(\sigma, N), \mathrm{T}_1\,\mathrm{M}\,p_1 = v_1;\ \mathrm{T}_2\,\mathrm{M}\,p_2 = v_2; \ldots;\ \mathrm{T}_n\,\mathrm{M}\,p_n = v_n;\ Block\rangle \to \langle(\sigma', R[v]), \cdot\rangle\end{array}}{\langle(\sigma, N), name(v_1, v_2, \ldots, v_n)\rangle \to \langle(\sigma', N), \cdot\rangle}$$

$$\textbf{TRANSITION-EXC} \quad \frac{\begin{array}{c}name \in \sigma_{functions}, \\ \Gamma_p\, name = Block, \qquad \Gamma_t\, name = ((p_1, \mathrm{T}_1), (p_2, \mathrm{T}_2), \ldots, (p_n, \mathrm{T}_n)) \\ \langle(\sigma, N), \mathrm{T}_1\,\mathrm{M}\,p_1 = v_1;\ \mathrm{T}_2\,\mathrm{M}\,p_2 = v_2; \ldots;\ \mathrm{T}_n\,\mathrm{M}\,p_n = v_n; Block\rangle \to \langle(\sigma', E), \cdot\rangle\end{array}}{\langle(\sigma, N), name(v_1, v_2, \ldots, v_n)\rangle \to \langle(\sigma', E), \cdot\rangle}$$

Note that, although the execution of the function ends with status $R[v]$ or $R[\cdot]$, the final status is N. Indeed, the function called has encountered a *return* statement, and therefore has finished, but the main execution is not necessarily finished yet.

When no function name matches the called function, Solidity provides a function, the *fallback function*, which will be invoked automatically. The fallback function does not have a name, it does not take any arguments and it does not return a value.

$$\textbf{TRANSITION-FAL} \quad \frac{\begin{array}{c}name \notin \sigma_{functions}, \\ \sigma_{fallback} = Block \\ \langle(\sigma, N), Block\rangle \to \langle(\sigma', R[\cdot]), \cdot\rangle\end{array}}{\langle(\sigma, N), name(v_1, v_2, ...)\rangle \to \langle(\sigma', N), \cdot\rangle}$$

$$\textbf{TRANSITION-FAL-EXC}\ \frac{\begin{array}{c} name \notin \sigma_{functions}, \\ \sigma_{fallback} = Block \\ \langle(\sigma', N), Block\rangle \rightarrow \langle(\sigma', E), \cdot\rangle \end{array}}{\langle(\sigma, N), name(v_1, v_2, ...)\rangle \rightarrow \langle(\sigma', E), \cdot\rangle}$$

Here we have used the symbol $\sigma_{fallback}$ as a way to represent the code assigned to the fallback function in the contract σ.

2.2.2 External function calls

Giving formal semantics to external function calls turns out to be a bit more intricate.

In order to do so, we need to introduce more notation:

- $\mathbb{A}$ will be a function mapping contract identifiers to their respective addresses.
- Δ is a function mapping contract addresses to their respective state σ.
- $\mathcal{C}$ is the set of all the contract identifiers defined in the blockchain.

Both $\mathbb{A}$ and Δ will be updated every time a new contract is deployed to the blockchain. Likewise, $\mathcal{C}$ will be extended so that it contains the name of the new contract.

The statement used for calling external functions is:

$$id.name.\text{value}(Exp_{eth}).\text{gas}(Exp_{gas})(Exp_1, Exp_2, \ldots, Exp_n),$$

where *id* is the name of a contract, *name* is the name of the function we want to call, the statement "value" is used to send Ether to the other contract, and the statement "gas" is used to set the amount of gas given to the contract to execute the function *name*. The rest of the expressions are the arguments given to the function.

We will try to make these rules as similar as possible to the ones given in Section 2.2.1. The first thing that is done is checking whether the contract name exists, and if the function's name has been declared. If it has been declared, then we evaluate each expression. If not, we will call the called contract's fallback function. Remember that the fallback function does not take any arguments.

$$\textbf{EF}_1\ \frac{\begin{array}{c} id \in \mathcal{C} \\ \mathbb{A}(id) = address, \qquad \Delta(address) = \sigma' \\ name \in \sigma'_{functions} \\ \text{Eval}(\sigma, Exp_{eth}) \rightarrow \langle(\sigma_1, N), v_{eth}\rangle \\ \text{Eval}(\sigma_1, Exp_{gas}) \rightarrow \langle(\sigma_2, N), v_{gas}\rangle \\ \text{Eval}(\sigma_2, Exp_1) \rightarrow \langle(\sigma_3, N), v_1\rangle \\ \text{Eval}(\sigma_3, Exp_2) \rightarrow \langle(\sigma_4, N), v_2\rangle \\ \vdots \\ \text{Eval}(\sigma_{n+1}, Exp_n) \rightarrow \langle(\sigma_{n+2}, N), v_n\rangle \\ \langle(\sigma_{n+2}, N), id.name.\text{value}(v_{eth}).\text{gas}(v_{gas})(v_1, v_2, \ldots, v_n)\rangle \rightarrow \langle(\sigma_{n+2}, N), \cdot\rangle \end{array}}{\langle(\sigma, N), \text{call } id.name.\text{value}(Exp_{eth}).\text{gas}(Exp_{gas})(Exp_1, Exp_2, \ldots, Exp_n)\rangle \rightarrow \langle(\sigma_{n+2}, N), \cdot\rangle}$$

$$\textbf{EF}_2\ \frac{\begin{array}{c} id \in \mathcal{C} \\ \mathbb{A}(id) = address, \qquad \Delta(address) = \sigma' \\ name \notin \sigma'_{functions} \\ \text{Eval}(\sigma, Exp_{eth}) \to \langle(\sigma_1, N), v_{eth}\rangle \\ \text{Eval}(\sigma_1, Exp_{gas}) \to \langle(\sigma_2, N), v_{gas}\rangle \\ \langle(\sigma_2, N), id.\text{fallback.value}(v_{eth}).\text{gas}(v_{gas})()\rangle \to \langle(\sigma_2, N), \cdot\rangle \end{array}}{\langle(\sigma, N), \text{call } id.name.\text{value}(Exp_{eth}).\text{gas}(Exp_{gas})(Exp_1, Exp_2, \ldots, Exp_n)\rangle \to\!\to \langle(\sigma_2, N), \cdot\rangle}$$

After evaluating every expression, we will execute the function itself. To do so, we first find the state that is associated to the contract, using $\mathbb{A}$ and Δ. Then we have to push the current state into the stack Ω. Finally, the code is executed. As before, we do not distinguish between functions that return a value or not. Note that the final state is not the final state after the execution of the function, this is because there has been a change in the environment of the execution.

$$\textbf{EXT-TRANS}\ \frac{\begin{array}{c} \mathbb{A}(id) = address, \qquad \Delta(address) = \sigma' \\ \Omega_{\sigma'} = \Omega_\sigma.\text{push}(\sigma) \\ \Gamma_p\, name = Block, \qquad \Gamma_t\, name = ((p_1, \text{T}_1), (p_2, \text{T}_2), \ldots, (p_n, \text{T}_n)) \\ \langle(\sigma', N), \text{T}_1\,\text{M}\,p_1 = v_1;\ \text{T}_2\,\text{M}\,p_2 = v_2; \ldots;\ \text{T}_n\,\text{M}\,p_n = v_n; Block\rangle \to \langle(\sigma'', R[\cdot]), \cdot\rangle \\ \Omega_{\sigma'} = \Omega_\sigma.\text{pop}() \end{array}}{\langle(\sigma, N), id.name.\text{value}(v_{eth}).\text{gas}(v_{gas})(v_1, v_2, \ldots, v_n)\rangle \to \langle(\sigma, N), \cdot\rangle}$$

$$\textbf{EXT-TRANS-RET}\ \frac{\begin{array}{c} \mathbb{A}(id) = address, \qquad \Delta(address) = \sigma' \\ \Omega_{\sigma'} = \Omega_\sigma.\text{push}(\sigma) \\ \Gamma_p\, name = Block, \qquad \Gamma_t\, name = ((p_1, \text{T}_1), (p_2, \text{T}_2), \ldots, (p_n, \text{T}_n)) \\ \langle(\sigma', N), \text{T}_1\,\text{M}\,p_1 = v_1;\ \text{T}_2\,\text{M}\,p_2 = v_2; \ldots;\ \text{T}_n\,\text{M}\,p_n = v_n; Block\rangle \to \langle(\sigma'', R[v]), \cdot\rangle \\ \Omega_{\sigma'} = \Omega_\sigma.\text{pop}() \end{array}}{\langle(\sigma, N), id.name.\text{value}(v_{eth}).\text{gas}(v_{gas})(v_1, v_2, \ldots, v_n)\rangle \to \langle(\sigma, N), \cdot\rangle}$$

$$\textbf{EXT-TRANS-EXC}\ \frac{\begin{array}{c} \mathbb{A}(id) = address, \qquad \Delta(address) = \sigma' \\ \Omega_{\sigma'} = \Omega_\sigma.\text{push}(\sigma) \\ \Gamma_p\, name = Block, \qquad \Gamma_t\, name = ((p_1, \text{T}_1), (p_2, \text{T}_2), \ldots, (p_n, \text{T}_n)) \\ \langle(\sigma', N), \text{T}_1\,\text{M}\,p_1 = v_1;\ \text{T}_2\,\text{M}\,p_2 = v_2; \ldots;\ \text{T}_n\,\text{M}\,p_n = v_n; Block\rangle \to \langle(\sigma'', E), \cdot\rangle \\ \Omega_{\sigma'} = \Omega_\sigma.\text{pop}() \end{array}}{\langle(\sigma, N), id.name.\text{value}(v_{eth}).\text{gas}(v_{gas})(v_1, v_2, \ldots, v_n)\rangle \to \langle(\sigma, E), \cdot\rangle}$$

$$\textbf{EXT-TRANS-FALL}\ \frac{\begin{array}{c} \mathbb{A}(id) = address, \qquad \Delta(address) = \sigma' \\ \Omega_{\sigma'} = \Omega_\sigma.\text{push}(\sigma) \\ \sigma'_{fallback} = Block \\ \langle(\sigma', N), Block\rangle \to \langle(\sigma'', R[\cdot]), \cdot\rangle \\ \Omega_{\sigma'} = \Omega_\sigma.\text{pop}() \end{array}}{\langle(\sigma, N), id.\text{fallback.value}(v_{eth}).\text{gas}(v_{gas})(v_1, v_2, \ldots, v_n)\rangle \to \langle(\sigma, N), \cdot\rangle}$$

$$\textbf{EXT-TRANS-FALL-EXC}\ \frac{\begin{array}{c}\mathbb{A}(id) = address, \qquad \Delta(address) = \sigma' \\ \Omega_{\sigma'} = \Omega_{\sigma}.\text{push}(\sigma) \\ \sigma'_{fallback} = Block \\ \langle(\sigma', N), Block\rangle \to \langle(\sigma'', E), \cdot\rangle \\ \Omega_{\sigma'} = \Omega_{\sigma}.\text{pop}()\end{array}}{\langle(\sigma, N), id.\text{fallback}.\text{value}(v_{eth}).\text{gas}(v_{gas})(v_1, v_2, \ldots, v_n)\rangle \to \langle(\sigma, E), \cdot\rangle}$$

2.3 Statements within the same function

In this section, we will provide rules for the most common statements.

When the execution status is E, $R[\cdot]$ or $R[v]$, it means we have previously encountered a *return* statement, or an exception has been triggered. Therefore, no further statement will be executed. The following rules take a statement and discard it, without changing the state or the execution status:

$$\textbf{SKIP-EXC}\ \frac{}{\langle(\sigma, E), Stmt\rangle \to \langle(\sigma, E), \cdot\rangle} \qquad \textbf{SKIP-RET}\ \frac{}{\langle(\sigma, R[v]), Stmt\rangle \to \langle(\sigma, R[v]), \cdot\rangle}$$

$$\textbf{SKIP-RET}\ \frac{}{\langle(\sigma, R[\cdot]), Stmt\rangle \to \langle(\sigma, R[\cdot]), \cdot\rangle}$$

If the statement to be executed is *return*, the execution status will be changed from N to $R[v]$ or $R[\cdot]$.

$$\textbf{RETURN}\ \frac{}{\langle(\sigma, N), \text{return}\rangle \to \langle(\sigma, R[\cdot]), \cdot\rangle} \qquad \textbf{RETURN-VAL}\ \frac{\text{Eval}(\sigma, Exp) \to \langle(\sigma', N), v\rangle}{\langle(\sigma, N), \text{return } Exp\rangle \to \langle(\sigma', R[v]), \cdot\rangle}$$

In case an exception is triggered while evaluating the expression, we will change the execution status.

$$\textbf{RETURN-EXC}\ \frac{\text{Eval}(\sigma, Exp) \to \langle(\sigma', E), \cdot\rangle}{\langle(\sigma, N), \text{return } Exp\rangle \to \langle(\sigma', E), \cdot\rangle}$$

For the remaining statements in this chapter, the rules are very similar to the rules that appear in semantics textbooks for simple, imperative languages. The composition rule is defined as expected:

$$\textbf{COMP}\ \frac{\langle(\sigma, N), Stmt_1\rangle \to \langle(\sigma_1, x_1), \cdot\rangle \quad \langle(\sigma_1, x_1), Stmt_2\rangle \to \langle(\sigma', x), \cdot\rangle}{\langle(\sigma, N), Stmt_1; Stmt_2\rangle \to \langle(\sigma', x), \cdot\rangle}$$

For the *while* statement we will give three different rules. We will be using one or the other depending on the outcome of the evaluation of the halt condition; the halt condition must evaluate to either true ($\top$), false ($\bot$), or raise an exception.

$$\textbf{WHILE}_1 \ \frac{\text{Eval}(\sigma, Exp) \to \langle(\sigma', N), \bot\rangle}{\langle(\sigma, N), \text{while}(Exp)\{Stmt\}\rangle \to \langle(\sigma', N), \cdot\rangle}$$

$$\textbf{WHILE}_2 \ \frac{\begin{array}{c}\text{Eval}(\sigma, Exp) \to \langle(\sigma', N), \top\rangle \\ \langle(\sigma', N), Stmt\rangle \to \langle(\sigma'', x_1), \cdot\rangle \\ \langle(\sigma'', x_1), \text{while}(Exp)\{Stmt\}\rangle \to \langle(\sigma''', x_2), \cdot\rangle\end{array}}{\langle(\sigma, N), \text{while}(Exp)\{Stmt\}\rangle \to \langle(\sigma''', x_2), \cdot\rangle}$$

$$\textbf{WHILE-EXC} \ \frac{\text{Eval}(\sigma, Exp) \to \langle(\sigma', E), v\rangle}{\langle(\sigma, N), \text{while}(Exp)\{Stmt\}\rangle \to \langle(\sigma', E), \cdot\rangle}$$

In the same manner as we gave three rules for the *while* loop, we now give three rules for the *if* statement, and three more for the *if-else* statement.

$$\textbf{IF}_1 \ \frac{\begin{array}{c}\text{Eval}(\sigma, Exp) \to \langle(\sigma', N), \top\rangle \\ \langle(\sigma', N), Stmt\rangle \to \langle(\sigma'', x), \cdot\rangle\end{array}}{\langle(\sigma, N), \text{if}(Exp)\{Stmt\}\rangle \to \langle(\sigma'', x), \cdot\rangle} \qquad \textbf{IF}_2 \ \frac{\text{Eval}(\sigma, Exp) \to \langle(\sigma', N), \bot\rangle}{\langle(\sigma, N), \text{if}(Exp)\{Stmt\}\rangle \to \langle(\sigma', N), \cdot\rangle}$$

$$\textbf{IF-EXC} \ \frac{\text{Eval}(\sigma, Exp) \to \langle(\sigma', E), v\rangle}{\langle(\sigma, N), \text{if}(Exp)\{Stmt\}\rangle \to \langle(\sigma', E), \cdot\rangle}$$

$$\textbf{IFELSE}_1 \ \frac{\begin{array}{c}\text{Eval}(\sigma, Exp) \to \langle(\sigma', N), \top\rangle \\ \langle(\sigma', N), Stmt_1\rangle \to \langle(\sigma'', x), \cdot\rangle\end{array}}{\langle(\sigma, N), \text{if}(Exp)\{Stmt_1\}\text{else}\{Stmt_2\}\rangle \to \langle(\sigma'', x), \cdot\rangle}$$

$$\textbf{IFELSE}_2 \ \frac{\begin{array}{c}\text{Eval}(\sigma, Exp) \to \langle(\sigma', N), \bot\rangle \\ \langle(\sigma', N), Stmt_2\rangle \to \langle(\sigma'', x), \cdot\rangle\end{array}}{\langle(\sigma, N), \text{if}(Exp)\{Stmt_1\}\text{else}\{Stmt_2\}\rangle \to \langle(\sigma'', x), \cdot\rangle}$$

$$\textbf{IFELSE-EXC} \ \frac{\text{Eval}(\sigma, Exp) \to \langle(\sigma', E), v\rangle}{\langle(\sigma, N), \text{if}(Exp)\{Stmt_1\}\text{else}\{Stmt_2\}\rangle \to \langle(\sigma', E), \cdot\rangle}$$

The last rule we provide will be used when an expression is encountered between statements, without being assigned. In this case, we simply evaluate the expression and continue with the execution.

$$\textbf{EXPRESSION} \ \frac{\text{Eval}(\sigma, Exp) \to \langle(\sigma', x), v\rangle}{\langle(\sigma, N), Exp\rangle \to \langle(\sigma', x), \cdot\rangle}$$

2.4 Declarations

In this chapter we will give semantics for variable declarations. When giving a value to a variable, we assume that there has been some kind of preprocessing to the code and thus expressions have the same type as the type declared. We could also ensure this by limiting assignments between different types in the syntax.

We need to follow four basic rules stated in the Solidity 0.6.4 documentation [11]:

1. Assignments between store and memory always create an individual copy.
2. Memory to memory assignments only create a reference.
3. Storage to local storage assignments only create a reference.
4. Other assignments to storage create an individual copy.

We will now provide some semantics for declaring variables both as state and local, according to the rules presented above.

2.4.1 Declaring state variables

State variables are always stored in storage. We also know that assignments between state variables always create a new copy.

Given an expression, we will evaluate it, save its type and address in our state, update the pointer Λ, and save the value in storage using $\mathcal{U}$, which will be studied in more depth in Section 3.1. $\mathcal{U}$ will take a configuration and a tuple containing the starting address from where the value has to be stored, the type of the value, and the value itself, and returns a new configuration in which the value has been stored.

$$\textbf{DEF-VAL-STR}_1 \quad \frac{\begin{array}{c}\text{Eval}(\sigma, \mathit{Exp}) \to \langle(\sigma', N), v\rangle \\ \mathit{id} \notin \mathbb{ID}_{\Psi_{\sigma'}} \\ \mathit{update\_state}(\sigma', \mathit{Type}, \mathit{id}) = \sigma'' \\ \mathcal{U}(\sigma'', (N_{\sigma''}\mathit{id}, \mathit{Type}, v)) \to \sigma'''\end{array}}{\langle(\sigma, N), \mathit{Type}\ \mathit{id} = \mathit{Exp}\rangle \to \langle(\sigma''', N), \cdot\rangle}$$

$$\textbf{DEF-VAL-STR}_1\textbf{-EXC} \quad \frac{\text{Eval}(\sigma, \mathit{Exp}) \to \langle(\sigma', E), v\rangle}{\langle(\sigma, N), \mathit{Type}\ \mathit{id} = \mathit{Exp}\rangle \to \langle(\sigma', E), \cdot\rangle}$$

The auxiliary function $\mathit{update\_state}(\sigma', \mathit{Type}, \mathit{id})$ is used to update the storage environment; given the current state, the type and the name for a new declaration, it returns an updated state where the functions τ, N and the parameter Λ have been modified accordingly.

Given the state $\sigma = (\Psi, M, \Omega)$, *update_state* will create a new state $\sigma' = (\Psi', M, \Omega)$ in which the new *id* has been added to the storage's name space for variables:

$$\mathbb{ID}_{\Psi'} = \mathbb{ID}_{\Psi} \cup \{\mathit{id}\}.$$

Also, the τ function is now defined over $\mathbb{ID}_{\Psi'}$:

$$\tau_{\Psi'} : \mathbb{ID}_{\Psi'} \to \mathbb{T}$$

and will return the following values:

$$\tau_{\Psi'}\, x = \begin{cases} \tau_{\Psi}\, x & x \neq id \\ \mathit{Type} & x = id \end{cases}.$$

We will modify the N function in a very similar way. It is now defined over $\mathbb{ID}_{\Psi'}$

$$N_{\Psi'} : \mathbb{ID}_{\Psi'} \to \mathbb{P}$$

and returns the following values:

$$N_{\Psi'}\, x = \begin{cases} N_{\Psi}\, x & x \neq id \\ \lceil \Lambda_{\sigma'} \rceil^{(l,\mathit{Type})} & x = id \end{cases}.$$

To finish, *update_state* updates the value of Λ so that it points to the next free address in the storage:

$$\Lambda_{\sigma'} = \Lambda_{\sigma} \uparrow (l, \mathit{Type}).$$

Now that we have seen in detail how this function works, we will use it in the following rules.

In case there is no value assigned to the variable, we will give it its default value, using DefVal, which will be presented in Section 5.6. DefVal maps every type declared to its default value.

$$\textbf{DEF-VAL-STR}_{\mathbf{2}}\ \frac{\begin{array}{c}\mathrm{DefVal}(\mathit{Type}) = v \\ id \notin \mathbb{ID}_{\Psi_\sigma} \\ \mathit{update\_state}(\sigma, \mathit{Type}, id) = \sigma' \\ \mathcal{U}(\sigma', (N_{\sigma''}\, id, \mathit{Type}, v)) \to \sigma''\end{array}}{\langle (\sigma, N), \mathit{Type}\ id \rangle \to \langle (\sigma'', N), \cdot \rangle}$$

The last rule is only used for dynamic arrays. Using the statement "new", a given size is set to the array. Each element in the array will be given its type's default value.

$$\textbf{DEF-NEW-DARRAY}\ \frac{\begin{array}{c} id \notin \mathbb{ID}_{\Psi_\sigma} \\ \mathrm{DefVal}(T) = v \\ \mathit{update\_state}(\sigma, T[], id) = \sigma' \\ \mathcal{U}(\sigma', (N_{\sigma'}\, id, T[], [v, v, \ldots^{n-1}, v])) \to \sigma''\end{array}}{\langle (\sigma, N), T[]\ id = \text{new}\ \ T(n) \rangle \to \langle (\sigma'', N), \cdot \rangle}$$

2.4.2 Declaring local variables

Now we will first focus on declaring local variables in storage and then in the memory. Please note that, unlike in the last section, where all variables where stored in memory, now there needs to appear an M or an S to state whether the variable will be saved in memory or storage.

We cannot declare mapping type variables within a function. We will not provide rules for mappings as we assume that they will not appear in our program since it has been preprocessed.

First of all, value type variables cannot be declared in storage as local variables.

$$\textbf{STORAGE-VT}\ \frac{\mathbb{T}(\mathit{Type}) = VT}{\langle(\sigma, N), \mathit{Type}\ S\ id = \mathit{Exp}\rangle \to \langle(\sigma, E), \cdot\rangle}$$

In case the expression which is being assigned to the declaration is a literal or is saved in the memory, we will create a new copy of its value in storage.

In the next rule, we first check that the type of the variable defined is a reference type using the function $\mathbb{T}$, next we check that the expression *Exp* is not located in memory or storage using the function $\mathcal{A}$. Both $\mathbb{T}$ and $\mathcal{A}$ will be discussed in Section 5. Finally, we evaluate *Exp* and follow the same steps we did earlier.

Some of the auxiliary functions that will appear are actually partial functions. When such functions take an input for which there is no result, the output will be an undefined value, represented by $\_$. In this case, $\mathcal{A}$ checks whether a function is located in memory or storage; if $\mathcal{A}$ receives a literal, it will return $\_$.

$$\textbf{STORAGE-LIT}\ \frac{\begin{array}{c}\mathbb{T}(\mathit{Type}) = RT \\ \mathcal{A}(\mathit{Exp}) = \_ \\ \mathrm{Eval}(\sigma, \mathit{Exp}) \to \langle(\sigma', N), v\rangle \\ id \notin \mathbb{ID}_{\Psi_{\sigma'}} \\ \mathit{update\_state}(\sigma', \mathit{Type}, id) = \sigma'' \\ \mathcal{U}(\sigma'', (N_{\sigma''} id, \mathit{Type}, v)) \to \sigma'''\end{array}}{\langle(\sigma, N), \mathit{Type}\ S\ id = \mathit{Exp}\rangle \to \langle(\sigma''', N), \cdot\rangle}$$

The following rule is almost the same as the last one, except that in this case we check that *Exp* is located in memory.

$$\textbf{STORAGE-MEM}\ \frac{\begin{array}{c}\mathbb{T}(\mathit{Type}) = RT \\ \mathcal{A}(\mathit{Exp}) = M \\ \mathit{retrieve\_mem}(\sigma, \mathit{Exp}) \to \langle N, v\rangle \\ id \notin \mathbb{ID}_{\Psi_{\sigma}} \\ \mathit{update\_state}(\sigma, \mathit{Type}, id) = \sigma' \\ \mathcal{U}(\sigma', (N_{\sigma'} id, \mathit{Type}, v)) \to \sigma''\end{array}}{\langle(\sigma, N), \mathit{Type}\ S\ id = \mathit{Exp}\rangle \to \langle(\sigma'', N), \cdot\rangle}$$

The function *retrieve_mem* is used to formalise the behaviour of the memory. Given

a state and an expression, it returns the value stored.

If the expression is stored in storage, then we only have to make our new variable point to the direction of the expression. To do so, we find where the expression is saved using the auxiliary function $\mathcal{D}$, which we will study in detail in Section 5. For now, the reader only needs to know that $\mathcal{D}$ takes a state, and an expression in which a variable is being accessed (for example, the element of an array, a member in a struct...), and returns an updated state, and the address that the expression is accessing.

$$\textbf{STORAGE-STR}\ \frac{\begin{array}{c}\mathbb{T}(Type) = RT \\ \mathcal{A}(Exp) = S \\ \mathcal{D}(\sigma, Exp) \to (\sigma', addr) \\ id \notin \mathbb{ID}_{\Psi_{\sigma'}} \\ update\_state\_pointer(\sigma', Type, id, addr) = \sigma''\end{array}}{\langle(\sigma, N), Type\ S\ id = Exp\rangle \to \langle(\sigma'', N), \cdot\rangle}$$

In this case we used $update\_state\_pointer(\sigma', Type, id, addr)$ instead of $update\_state$. The reason for this is that we are not interested in updating the pointer Λ, and now we want the value $N\,x$ to be a certain address, not the default value we gave in the previous rules.

Given the state $\sigma = (\Psi, M, \Omega)$, $update\_state\_pointer$ will create a new state $\sigma' = (\Psi', M, \Omega)$, where $\mathbb{ID}_{\Psi'} = \mathbb{ID} \cup \{id\}$. Like before, we have

$$\tau_{\Psi'} : \mathbb{ID}_{\Psi'} \to \mathbb{T}$$

$$\tau_{\Psi'} x = \begin{cases} \tau_\Psi x & x \neq id \\ Type & x = id \end{cases},$$

and

$$N_{\Psi'} : \mathbb{ID}_{\Psi'} \to \mathbb{P}$$

$$N_{\Psi'} x = \begin{cases} N_\Psi x & x \neq id \\ addr & x = id \end{cases}.$$

Local variables declared in storage are always reference types. If no expression is given, they will be assigned direction 0, acting like a pointer. This may cause strange executions, due to unwanted modifications in storage addresses, and therefore it should be avoided.

$$\textbf{STORAGE}\ \frac{\begin{array}{c}\mathbb{T}(\mathit{Type}) = \mathit{RT} \\ \mathit{id} \notin \mathbb{ID}_{\Psi_\sigma} \\ \mathit{update\_state\_pointer}(\sigma, \mathit{Type}, \mathit{id}, 0) = \sigma'\end{array}}{\langle(\sigma, N),\ \mathit{Type}\ S\ \mathit{id}\rangle \to \langle(\sigma', N), \cdot\rangle}$$

The next rule is identical to DEF-NEW-DARRAY (Section 2.4.1). Therefore, no further explanation is needed.

$$\textbf{STORAGE-NEW-DARRAY}\ \frac{\begin{array}{c}\mathit{id} \notin \mathbb{ID}_{\Psi_\sigma} \\ \mathrm{DefVal}(T) = v \\ \mathit{update\_state}(\sigma, T[], \mathit{id}) = \sigma' \\ \mathcal{U}(\sigma', (N_{\sigma'}\ \mathit{id}, T[], [v, v, \ldots^{n-1}, v])) \to \sigma''\end{array}}{\langle(\sigma, N),\ T[]\ S\ \mathit{id} = \text{new}\ T(n)\rangle \to \langle(\sigma'', N), \cdot\rangle}$$

Now we present rules for declaring memory variables. As before, we give three different rules: one to apply when the variable is not being initialised, and two more, depending on where the expression is located.

If the expression is either a literal or it is located in storage, a copy is stored.

$$\textbf{MEM-OTHR}\ \frac{\begin{array}{c}\mathcal{A}(\mathit{Exp}) = \_ \ \vee\ \mathcal{A}(\mathit{Exp}) = S \\ \mathrm{Eval}(\sigma, \mathit{Exp}) \to \langle(\sigma', N), v\rangle \\ \mathit{id} \notin \mathbb{ID}_{M_{\sigma'}} \\ \mathit{emplace\_mem}(\sigma', \mathit{id}, \mathit{Type}, v) = \sigma''\end{array}}{\langle(\sigma, N),\ \mathit{Type}\ M\ \mathit{id} = \mathit{Exp}\rangle \to \langle(\sigma'', N), \cdot\rangle}$$

On the other hand, if the expression is located in memory, then only a pointer will be created in memory.

$$\textbf{MEM-MEM}\ \frac{\begin{array}{c}\mathcal{A}(\mathit{Exp}) = M \\ \mathit{id} \notin \mathbb{ID}_{M_\sigma} \\ \mathit{ref\_mem}(\sigma, \mathit{id}, \mathit{Exp}) = \sigma'\end{array}}{\langle(\sigma, N),\ \mathit{Type}\ M\ \mathit{id} = \mathit{Exp}\rangle \to \langle(\sigma', N), \cdot\rangle}$$

The rule used to declare variables without giving them an initial value is quite straightforward; instead of evaluating an expression, we get the default value for the variable using the function DefVal.

$$\textbf{MEM}\ \frac{\begin{array}{c}\mathit{id} \notin \mathbb{ID}_{M_\sigma} \\ \mathrm{DefVal}(T) = v \\ \mathit{emplace\_mem}(\sigma, \mathit{id}, \mathit{Type}, v) = \sigma'\end{array}}{\langle(\sigma, N),\ \mathit{Type}\ M\ \mathit{id}\rangle \to \langle(\sigma', N), \cdot\rangle}$$

To finish this section, we give a rule very similar to STORAGE-NEW-DARRAY. The only difference is that now the array will be stored in memory.

$$\textbf{MEM-NEW-DARRAY}\ \frac{\begin{array}{c} id \notin \mathbb{ID}_{M_\sigma} \\ \text{DefVal}(T) = v \\ \mathit{emplace\_mem}(\sigma, id, T[], [v, v, \ldots^{n-1)}, v]) = \sigma' \end{array}}{\langle(\sigma, N), T[]\ M\ id = \text{new}\ T(n)\rangle \rightarrow \langle(\sigma'', N), \cdot\rangle}$$

We have used three auxiliary functions, which we will not describe in depth, as the memory model in Solidity turns out to be somehow confusing. We have used *retrieve_mem*, which, given a state and an expression, accesses the memory address where that expression is stored and returns the value. *Emplace_mem* updates the memory given a new variable identifier, its type, and the value we want to give it. Lastly, *ref_mem* is used to update the value to which a certain variable is pointing, changing it to the address pointed by the expression it takes as an argument.

The three auxiliary memory functions described above also modify τ_M, N_M, and $\mathbb{ID}_M$ in a very similar way to how *update_state* and *update_state_pointer* did. In this case, we will not give more detail.

2.5 Assignments

These rules are quite similar to those presented in Section 2.4.2. Therefore, we will not describe them as much as we did then.

The first thing that has to be checked when executing an assignment is that the expression on the left must we something we can assign a value to. In this manner, if the expression on the left is a literal, an exception will be raised.

The auxiliary function $\mathcal{T}$ is used to check the type of the value pointed by an expression. It will be studied in Section 5.1.

$$\textbf{ASS-EXC}\ \frac{\mathcal{T}(Exp_1) = \_ \ \vee\ \mathcal{T}(Exp_1) = [n]\ \vee\ \mathcal{T}(Exp_1) = \mathit{struct\_lit}\ \vee\ \mathcal{T}(Exp_1) = \mathit{map\_lit}}{\langle(\sigma, N), Exp_1 = Exp_2\rangle \rightarrow \langle(\sigma, E), \cdot\rangle}$$

As before, we will assume that both expressions have the same type.

This first rule will be used to assign a literal expression to a variable in storage. In order to do so, we need to evaluate the literal expression, find where Exp_1 is located in storage using $\mathcal{D}$, and then just update the value in storage using $\mathcal{U}$.

$$\textbf{ASS-STR-LIT}\ \frac{\begin{array}{c} \mathcal{T}(Exp_1) = \text{T} \\ \mathcal{A}(Exp_1) = S, \qquad \mathcal{A}(Exp_2) = \_ \\ \text{Eval}(\sigma, Exp_2) \rightarrow \langle(\sigma', N), v\rangle \\ \mathcal{D}(\sigma', Exp_1) \rightarrow (\sigma'', addr) \\ \mathcal{U}(\sigma'', (addr, \text{T}, v)) \rightarrow \sigma''' \end{array}}{\langle(\sigma, N), Exp_1 = Exp_2\rangle \rightarrow \langle(\sigma''', N), \cdot\rangle}$$

If the expression on the right hand side is located in memory, we will do the same thing as before, but instead of evaluating the Exp_2, we retrieve its value.

$$\textbf{ASS-STR-MEM}\ \frac{\begin{array}{c}\mathcal{T}(Exp_1) = \mathrm{T} \\ \mathcal{A}(Exp_1) = S, \qquad \mathcal{A}(Exp_2) = M \\ retrieve\_mem(\sigma, Exp_2) \to \langle N, v\rangle \\ \mathcal{D}(\sigma, Exp_1) \to (\sigma', addr) \\ \mathcal{U}(\sigma', (addr, \mathrm{T}, v)) \to \sigma''\end{array}}{\langle(\sigma, N), Exp_1 = Exp_2\rangle \to \langle(\sigma'', N), \cdot\rangle}$$

When assigning expressions located in storage, we need to check whether the expression on the left hand side is a state variable or not; we will do so using the auxiliary function $\mathcal{L}$, which we will study in Section 5.3. If it is, the value on the right hand side will be copied to the variable on the left hand side.

$$\textbf{ASS-STR-STR}_1\ \frac{\begin{array}{c}\mathcal{T}(Exp_1) = \mathrm{T} \\ \mathcal{A}(Exp_1) = S, \qquad \mathcal{A}(Exp_2) = S \\ \mathcal{L}(Exp_1) = ST \\ \mathrm{Eval}(\sigma, Exp_2) \to \langle(\sigma', N), v\rangle \\ \mathcal{D}(\sigma', Exp_1) \to (\sigma'', addr) \\ \mathcal{U}(\sigma'', (addr, \mathrm{T}, v)) \to \sigma'''\end{array}}{\langle(\sigma, N), Exp_1 = Exp_2\rangle \to \langle(\sigma''', N), \cdot\rangle}$$

If we are assigning variables to a variable located in storage that is not a state variable, then only the address of the variable on the left hand side will change, *i.e.* the variable on the left hand side acts like a pointer. Note that this operation is only allowed when the expression on the left hand side is an identifier; it would result in many errors if different members of the same variable could point to different addresses; for example, a struct having each of its members stored in non-sequential addresses, and with no method to find where each member is stored.

$$\textbf{ASS-STR-STR}_2\ \frac{\begin{array}{c}\mathcal{T}(id) = \mathrm{T} \\ \mathcal{A}(id) = S, \qquad \mathcal{A}(Exp_2) = S \\ id \in \mathbb{ID}_\Psi, \qquad \mathcal{L}(id) = L \\ \mathcal{D}(\sigma, Exp_2) \to (\sigma', addr) \\ update\_state\_pointer(\sigma', \mathrm{T}, id, addr) = \sigma'\end{array}}{\langle(\sigma, N), id = Exp_2\rangle \to \langle(\sigma'', N), \cdot\rangle}$$

After presenting assignment rules for a variable located in storage, we will present two rules for assignments of variables when the one on the left hand side is located in memory.

As before, we will use auxiliary functions that will help us define operations in the memory. In this case, we will use *update_mem*, which updates the value stored in *Exp_1*'s address to v, and *update_ref_mem*, which updates Exp_1's pointer, so that it points to Exp_2.

If the expression on the right hand side is either a literal or located in storage, the value will be copied into memory.

$$\textbf{ASS-MEM-OTHR}\ \frac{\begin{array}{c}\mathcal{A}(Exp_1) = M \\ \mathcal{A}(Exp_2) = \_ \ \vee\ \mathcal{A}(Exp_2) = S \\ \text{Eval}(\sigma, Exp_2) \to \langle(\sigma', N), v\rangle \\ update\_mem(\sigma', Exp_1, v) = \sigma''\end{array}}{\langle(\sigma, N), Exp_1 = Exp_2\rangle \to \langle(\sigma'', N), \cdot\rangle}$$

In case both expressions are located in memory, then only a change in the address pointed by the expression on the left hand side will happen.

$$\textbf{ASS-MEM-MEM}\ \frac{\begin{array}{c}\mathcal{A}(Exp_1) = M, \qquad \mathcal{A}(Exp_2) = M \\ update\_ref\_mem(\sigma, Exp_1, Exp_2) = \sigma'\end{array}}{\langle(\sigma, N), Exp_1 = Exp_2\rangle \to \langle(\sigma', N), \cdot\rangle}$$

2.6 Events

The following statements are used to manage events. Events are used in Solidity to notify actions; they take a list of arguments and log their values in the blockchain.

Events are called using the word "emit", followed by the name of the event. To call an event, it has to be declared within the scope of the contract. We will check this by demanding $name \in \sigma_{events}$.

Once an event is called, each argument given to it will be evaluated, and the results will be written in the ledger using the auxiliary function "Log". The Log function can be specified *ad hoc* when implementing the semantics in a language like K or Maude, as its behaviour will depend on how the ledger is represented.

$$\textbf{EVENT}\ \frac{\begin{array}{c}name \in \sigma_{events} \\ \text{Eval}(\sigma, Exp_1) \to \langle(\sigma_1, N), v_1\rangle \\ \text{Eval}(\sigma_1, Exp_2) \to \langle(\sigma_2, N), v_2\rangle \\ \vdots \\ \text{Eval}(\sigma_{n-1}, Exp_n) \to \langle(\sigma_n, N), v_n\rangle \\ \text{Log}(\sigma_n, (name, v_1, v_2, \ldots, v_n)) \to (\sigma', N)\end{array}}{\langle(\sigma, N), \text{emit } name(Exp_1, Exp_2, \ldots, Exp_n)\rangle \to \langle(\sigma', N), \cdot\rangle}$$

$$\textbf{EVENT-EXC}\ \frac{\begin{array}{c}name \in \sigma_{events} \\ \text{Eval}(\sigma, Exp_1) \to \langle(\sigma_1, x_1), v_1\rangle \\ \text{Eval}(\sigma_1, Exp_2) \to \langle(\sigma_2, x_2), v_2\rangle \\ \vdots \\ \text{Eval}(\sigma_{n-1}, Exp_n) \to \langle(\sigma_n, x_n), v_n\rangle \\ \text{Log}(\sigma_n, (name, v_1, v_2, \ldots, v_n)) \to (\sigma', y) \\ x_1 = E \vee x_2 = E \vee \cdots \vee x_n = E \vee y = E\end{array}}{\langle(\sigma, N), \text{emit } name(Exp_1, Exp_2, \ldots, Exp_n)\rangle \to \langle(\sigma', E), \cdot\rangle}$$

2.7 Arrays

In this last section we consider some expressions used to modify arrays. We will only give semantics for arrays stored in storage, as giving semantics for arrays located in memory would not give any additional information.

The first statement we will look at is *push*, used to add an element at the end of a dinamic array. There are two variants of this operation: $push()$ and $push(Exp)$.

Using the statement $push()$, a new element with the default value for the type of the array will be added at the end of it. To do so, we use DefVal to get the default value of the type, then we get the address where the expression is pointing, update the size of the array, and save the new value in the correct address.

To find the correct address we use the function *shift*, which will be introduced in the next chapter. For now it is enough to know that $shift(addr, n, \mathrm{T})$ takes the address of an array, and returns the address of the nth element, provided that the type of the array is T.

$$\textbf{PUSH}_1 \ \frac{\begin{array}{c}\mathcal{A}(Exp) = S, \qquad \mathcal{T}(Exp) = \mathrm{T}[] \\ \mathrm{DefVal}(\mathrm{T}) = v \\ \mathcal{D}(\sigma, Exp) \to (\sigma', addr) \\ [addr]^{\mathrm{Size}(uint)}_{S_{\sigma'}} = n \\ [addr]^{\mathrm{Size}(uint)}_{S''_{\sigma}} = n + 1 \\ \mathcal{U}(\sigma'', (shift(\mathbf{HASH}(addr), n, \mathrm{T}), \mathrm{T}, v)) \to \sigma'''\end{array}}{\langle (\sigma, N), Exp.\mathrm{push}() \rangle \to \langle (\sigma''', N), \cdot \rangle}$$

We have used $\mathbf{HASH}(v)$ to denote the function that, given a value v, computes its hash.

If we provide an expression to the statement *push*, then a new element with the value of Exp_2 will be added to the array.

$$\textbf{PUSH}_2 \ \frac{\begin{array}{c}\mathcal{A}(Exp) = S \\ \mathcal{T}(Exp_1) = \mathrm{T}[], \qquad \mathcal{T}(Exp_2) = \mathrm{T} \ \vee \ \mathcal{T}(Exp_2) = \_ \\ \mathrm{Eval}(\sigma, Exp_2) \to \langle (\sigma_1, N), v \rangle \\ \mathcal{D}(\sigma_1, Exp) \to (\sigma_2, addr) \\ [addr]^{\mathrm{Size}(uint)}_{S_{\sigma_2}} = n \\ [addr]^{\mathrm{Size}(uint)}_{S_{\sigma_3}} = n + 1 \\ \mathcal{U}(\sigma_3, (shift(\mathbf{HASH}(addr), n, \mathrm{T}), \mathrm{T}, v)) \to \sigma_4\end{array}}{\langle (\sigma, N), Exp_1.\mathrm{push}(Exp_2) \rangle \to \langle (\sigma_4, N), \cdot \rangle}$$

The *pop* statement deletes the last element of a given array. This is simply done by reducing the size of the array.

$$\textbf{POP}\ \frac{\begin{array}{c}\mathcal{A}(Exp) = S, \qquad \mathcal{T}(Exp) = \mathrm{T}[] \\ \mathcal{D}(\sigma, Exp) \to (\sigma', addr) \\ [addr]_{S_{\sigma'}}^{\mathrm{Size}(address)} > 0 \\ [addr]_{S_{\sigma''}}^{\mathrm{Size}(address)} = [addr]_{S_{\sigma'}}^{\mathrm{Size}(address)} - 1\end{array}}{\langle(\sigma, N), Exp.\mathrm{pop}()\rangle \to \langle(\sigma'', N), \cdot\rangle}$$

3 Storage

This chapter is probably one of the most important in this document. Now we will focus on giving semantics to access and update the storage. Not only this is something we have not found in any paper we have consulted, but it also provides rules that will be used in almost every other section.

The first thing that has to be taken into account is the rules provided in Solidity's documentation [11] :

- The first item in a storage slot is stored lower-order aligned.
- Elementary types use only as many bytes as are necessary to store them.
- If an elementary type does not fit in the remaining part of a storage slot, it is moved to the next storage slot.
- Structs and array data always start a new slot and occupy whole slots (but items inside a struct or array are packed tightly according to these rules).

3.1 Updating the storage

We start with the rules that are used to update the storage. We define $\mathcal{U}$, which takes the starting address where a value will be stored, its type, and the value itself as arguments, and results in a new updated state, containing the new value.

If T is a simple type, then we only have to save its value in the given address, as shown.

$$\textbf{UPDATE-VALUE}\ \frac{\begin{array}{c}\mathrm{T} \in PrimType \\ [addr]_{S_\sigma}^{\mathrm{Size}(T)} = v\end{array}}{\mathcal{U}(\sigma, (addr, \mathrm{T}, v)) \to \sigma'}$$

If the type is not simple, we use $\mathcal{U}$ recursively to store the value of each of the members. Keep in mind that types can also be recursive, and therefore it is necessary to define the rules taking into consideration that the address must be aligned so that the values are stored according to the rules. To do so, we use the auxiliary function *shift*:

$$\mathit{shift}(\mathit{addr}, n, \mathrm{T}) = \begin{cases} \lceil \mathit{addr} \rceil^{(l,\mathrm{T})} & n = 0 \\ \lceil \mathit{shift}(\mathit{addr}, n-1, \mathrm{T}) + \mathrm{Size}(\mathrm{T}) \rceil^{(l,\mathrm{T})} & n > 0 \end{cases}.$$

The simplest types to store are arrays and structs; in both cases, each of their elements are stored consecutively.

$$\textbf{UPDATE-SARRAY}\ \frac{\begin{array}{c}\mathcal{U}(\sigma, (\mathit{addr}, \mathrm{T}, v_1)) \to \sigma_1 \\ \mathcal{U}(\sigma_1, (\mathit{shift}(\mathit{addr}, 1, \mathrm{T}), \mathrm{T}, v_2)) \to \sigma_2 \\ \mathcal{U}(\sigma_2, (\mathit{shift}(\mathit{addr}, 2, \mathrm{T}), \mathrm{T}, v_3)) \to \sigma_3 \\ \vdots \\ \mathcal{U}(\sigma_{n-1}, (\mathit{shift}(\mathit{addr}, n-1, \mathrm{T}), \mathrm{T}, v_n)) \to \sigma_n \end{array}}{\mathcal{U}(\sigma, (\mathit{addr}, \mathrm{T}[n], [v_1, v_2, \ldots, v_n])) \to \sigma_n}$$

Dynamic arrays are stored almost in the same way as static arrays. The only difference is that we need to compute the hash code of the address where the size of the array is stored.

$$\textbf{UPDATE-DARRAY}\ \frac{\begin{array}{c}[\mathit{addr}]_{S_\sigma}^{\mathrm{Size}(\mathit{uint})} = n \\ \mathit{addr}_2 = \lceil \mathbf{HASH}(\mathit{addr}) \rceil^{(l,\mathrm{T})} \\ \mathcal{U}(\sigma, (\mathit{addr}_2, \mathrm{T}, v_1)) \to \sigma_1 \\ \mathcal{U}(\sigma_1, (\mathit{shift}(\mathrm{addr}_2, 1, \mathrm{T}), \mathrm{T}, v_2)) \to \sigma_2 \\ \vdots \\ \mathcal{U}(\sigma_{n-1}, (\mathit{shift}(\mathrm{addr}_2, n-1, \mathrm{T}), \mathrm{T}, v_n)) \to \sigma_n \end{array}}{\mathcal{U}(\sigma, (\mathit{addr}, \mathrm{T}[], [v_1, v_2, \ldots, v_n])) \to \sigma_n}$$

When storing structs, things get a little bit more intricate, as each member of the struct has its own type. We first need to know its members and the members' types, using $\mathcal{S}$, which was introduced in Section 1.7. Then, we recursively use $\mathcal{U}$ to store each value.

$$\textbf{UPDATE-STRUCT}\ \frac{\begin{array}{c}\mathrm{T} \in \mathbb{S} \\ \mathcal{S}(\mathrm{T}) = \langle (id_1, \mathrm{T}_1), (id_2, \mathrm{T}_2), \ldots, (id_n, \mathrm{T}_n) \rangle \\ \mathcal{U}(\sigma, (\mathit{addr}, \mathrm{T}_1, v_1)) \to \sigma_1 \\ \mathcal{U}(\sigma_1, (\mathcal{O}(\mathit{addr}, id_2, \mathrm{T}), \mathrm{T}_2, v_2)) \to \sigma_2 \\ \mathcal{U}(\sigma_2, (\mathcal{O}(\mathit{addr}, id_3, \mathrm{T}), \mathrm{T}_3, v_3)) \to \sigma_3 \\ \vdots \\ \mathcal{U}(\sigma_{n-1}, (\mathcal{O}(\mathit{addr}, id_n, \mathrm{T}), \mathrm{T}_n, v_n)) \to \sigma_n \end{array}}{\mathcal{U}(\sigma, (\mathit{addr}, \mathrm{T}, [v_1, v_2, \ldots, v_n])) \to \sigma_n}$$

We have used the function $\mathcal{O}$, which does something similar to what *shift* does, but takes different arguments, to make notation easier. This function will be studied in detail in Section 5.5.

Lastly, we provide the rule to store values in a map. Here, the expression $\mathit{addr} \cdot c_i$

means the address concatenated to the value c_i

$$\textbf{UPDATE-MAP}\ \frac{\begin{array}{c}\mathcal{U}(\sigma, (\lceil \textbf{HASH}(addr \cdot c_i)\rceil^{(l,\mathrm{T})}, \mathrm{T}, v_i)) \to \sigma_1 \\ \mathcal{U}(\sigma_1, (\lceil \textbf{HASH}(addr \cdot c_i)\rceil^{(l,\mathrm{T})}, \mathrm{T}, v_i)) \to \sigma_2 \\ \vdots \\ \mathcal{U}(\sigma_{n-1}, (\lceil \textbf{HASH}(addr \cdot c_i)\rceil^{(l,\mathrm{T})}, \mathrm{T}, v_i)) \to \sigma_n\end{array}}{\mathcal{U}(\sigma, (addr, \text{mapping } (\mathrm{K} => \mathrm{T}), [\{c_1 : v_1\}, \{c_2 : v_2\}, \ldots, \{c_n : v_n\}])) \to \sigma_n}$$

As the reader may have already noticed, using these rules the value of the pointer Λ remains unchanged. This is because the rules presented above can be used to either declare new variables or to update the value of variables already declared.

3.2 Getting values from storage

The following rules are very similar to the rules given to update the storage in Section 3.1. But now, instead of saving values in the storage, we retrieve the information stored, given the contract state, an address, and the type of the value to be retrieved.

As before, if the type of the variable to be retrieved is simple, then the storage is accessed.

$$\textbf{GET-VALUE}\ \frac{\begin{array}{c}\mathrm{T} \in PrimType \\ [addr]_{S_\sigma}^{\mathrm{Size}(T)} = v\end{array}}{\mathcal{G}(\sigma, (addr, \mathrm{T})) \to v}$$

The rules we created for arrays and structs are so similar to the ones given to update the storage that we believe that they are self-explanatory.

$$\textbf{GET-SARRAY}\ \frac{\begin{array}{c}\mathcal{G}(\sigma, (addr, \mathrm{T})) \to v_1 \\ \mathcal{G}(\sigma, (shift(addr, 1, \mathrm{T}), \mathrm{T})) \to v_2 \\ \vdots \\ \mathcal{G}(\sigma, (shift(addr, n-1, \mathrm{T}), \mathrm{T})) \to v_n\end{array}}{\mathcal{G}(\sigma, (addr, \mathrm{T}[n])) \to [v_1, v_2, \ldots, v_n]}$$

$$\textbf{GET-DARRAY}\ \frac{\begin{array}{c}[addr]_{S_\sigma}^{\mathrm{Size}(uint)} = n \\ addr_2 = \lceil \textbf{HASH}(addr)\rceil^{(l,\mathrm{T})} \\ \mathcal{G}(\sigma, (addr_2, \mathrm{T})) \to v_1 \\ \mathcal{G}(\sigma, (shift(addr_2, 1, \mathrm{T}), \mathrm{T})) \to v_2 \\ \vdots \\ \mathcal{G}(\sigma, (shift(addr_2, n-1, \mathrm{T}), \mathrm{T})) \to v_n\end{array}}{\mathcal{G}(\sigma, (addr, \mathrm{T}[])) \to [v_1, v_2, \ldots, v_n]}$$

$$\textbf{GET-STRUCT}\ \frac{\begin{array}{c}\mathrm{T} \in \mathbb{S} \\ \mathcal{S}(\mathrm{T}) = \langle (id_1, \mathrm{T}_1), (id_2, \mathrm{T}_2), \ldots, (id_n, \mathrm{T}_n) \rangle \\ \mathcal{G}(\sigma, (addr, \mathrm{T}_1)) \to v_1 \\ \mathcal{G}(\sigma, (\mathcal{O}(addr, id_2, \mathrm{T}), \mathrm{T}_2)) \to v_2 \\ \vdots \\ \mathcal{G}(\sigma, (\mathcal{O}(addr, id_n, \mathrm{T}), \mathrm{T}_n)) \to v_n\end{array}}{\mathcal{G}(\sigma, (addr, \mathrm{T})) \to [v_1, v_2, \ldots, v_n]}$$

To get the value of a mapping, given its address, we need to know the keys stored in it. Even though this is not provided in Solidity, we decided to include it, to allow us to formalise it. We will use the function $\mathcal{M}$ from Section 1.7 for this purpose. Once we have the keys, we call $\mathcal{G}$ recursively to get each value.

$$\textbf{GET-MAP}\ \frac{\begin{array}{c}\mathcal{M}(N^{-1}addr) = (c_1, c_2, \ldots, c_n) \\ \mathcal{G}(\sigma, (\lceil \mathbf{HASH}(addr \cdot c_1) \rceil^{(l,\mathrm{T})}, \mathrm{T})) \to v_1 \\ \mathcal{G}(\sigma, (\lceil \mathbf{HASH}(addr \cdot c_2) \rceil^{(l,\mathrm{T})}, \mathrm{T})) \to v_2 \\ \vdots \\ \mathcal{G}(\sigma, (\lceil \mathbf{HASH}(addr \cdot c_n) \rceil^{(l,\mathrm{T})}, \mathrm{T})) \to v_n\end{array}}{\mathcal{G}(\sigma, (addr, \text{mapping } (\mathrm{K} => \mathrm{T}))) \to [\{c_1 : v_1\}, \{c_2 : v_2\}, \ldots, \{c_n : v_n\}]}$$

4 Evaluations

Along with the previous chapter, this is also one of the most important and intricate in this work. Many of the rules did not appear in any of the papers we have read, this is why we believe it is important to provide them.

4.1 Evaluating function calls

The first thing we will address is the evaluation of function calls. The following rules are very similar to those presented in Section 2.2. Therefore, we may not give as many details as we gave then.

4.1.1 Internal function calls

In this case, we will also divide our rules into two groups. Some of them will take

$$\text{e-call } name(Exp_1, Exp_2, \ldots, Exp_n)$$

as their argument, and others will only take the function name and a value for each argument, without the "e-call" statement. We have decided to do such division to simplify the notation used.

We use "e-call" and "call" as a way to distinguish function calls when we want to interpret them as expressions or statements. As we have seen before, if we want to interpret a function call as a statement, we will use "call", and when we want to interpret it as an expression, we will use "e-call".

The first thing that is done is evaluating each of the arguments, and then the function itself is evaluated.

$$\textbf{EVAL-FUN-CALL}\ \frac{\begin{array}{c}\text{Eval}(\sigma, \mathit{Exp}_1) \to \langle(\sigma_1, N), v_1\rangle \\ \text{Eval}(\sigma_1, \mathit{Exp}_2) \to \langle(\sigma_2, N), v_2\rangle \\ \vdots \\ \text{Eval}(\sigma_{n-1}, \mathit{Exp}_n) \to \langle(\sigma_n, N), v_n\rangle \\ \text{Eval}(\sigma_n, \mathit{name}(v_1, v_2, \ldots v_n)) \to \langle(\sigma', x), v\rangle\end{array}}{\text{Eval}(\sigma, \text{e-call } \mathit{name}(\mathit{Exp}_1, \mathit{Exp}_2, \ldots, \mathit{Exp}_n)) \to \langle(\sigma', x), v\rangle}$$

$$\textbf{EVAL-FUN-CALL-EXC}\ \frac{\begin{array}{c}\text{Eval}(\sigma, \mathit{Exp}_1) \to \langle(\sigma_1, x_1), v_1\rangle \\ \text{Eval}(\sigma_1, \mathit{Exp}_2) \to \langle(\sigma_2, x_2), v_2\rangle \\ \vdots \\ \text{Eval}(\sigma_{n-1}, \mathit{Exp}_n) \to \langle(\sigma_n, x_n), v_n\rangle \\ x_1 = E \ \vee x_2 = E \ \vee \cdots \vee x_n = E\end{array}}{\text{Eval}(\sigma, \text{e-call } \mathit{name}(\mathit{Exp}_1, \mathit{Exp}_2, \ldots, \mathit{Exp}_n)) \to \langle(\sigma_n, E), \cdot\rangle}$$

The following rules will be used to evaluate the function itself. This first rule is almost identical to TRANSITION-RET (Section 2.2.1):

$$\textbf{EVAL-FUN}\ \frac{\begin{array}{c}\mathit{name} \in \sigma_{\mathit{functions}}, \\ \Gamma_p\, \mathit{name} = \mathit{Block}, \qquad \Gamma_t\, \mathit{name} = ((p_1, \mathrm{T}_1), (p_2, \mathrm{T}_2), \ldots, (p_n, \mathrm{T}_n)) \\ \langle(\sigma, N), \mathrm{T}_1\, M\, p_1 = v_1;\ \mathrm{T}_2\, M\, p_2 = v_2; \ldots;\ \mathrm{T}_n\, M\, p_n = v_n;\ \mathit{Block}\rangle \to \langle(\sigma', R[v]), \cdot\rangle\end{array}}{\text{Eval}(\sigma, \mathit{name}(v_1, v_2, \ldots, v_n)) \to \langle(\sigma', N), v\rangle}$$

In case the function called does not return any value, an exception will be raised; we always expect values when evaluating.

$$\textbf{EVAL-FUN-EXC}\ \frac{\begin{array}{c}\mathit{name} \in \sigma_{\mathit{functions}}, \\ \Gamma_p\, \mathit{name} = \mathit{Block}, \qquad \Gamma_t\, \mathit{name} = ((p_1, \mathrm{T}_1), (p_2, \mathrm{T}_2), \ldots, (p_n, \mathrm{T}_n)) \\ \langle(\sigma, N), \mathrm{T}_1\, M\, p_1 = v_1;\ \mathrm{T}_2\, M\, p_2 = v_2; \ldots;\ \mathrm{T}_n\, M\, p_n = v_n; \mathit{Block}\rangle \to \langle(\sigma', R[\cdot]), \cdot\rangle\end{array}}{\text{Eval}(\sigma, \mathit{name}(v_1, v_2, \ldots, v_n)) \to \langle(\sigma', E), \cdot\rangle}$$

The same thing will happen if the fallback function is executed.

$$\textbf{EVAL-FUN-FAL}\ \frac{\begin{array}{c}\mathit{name} \notin \sigma_{\mathit{functions}}, \\ \sigma_{\mathit{fallback}} = \mathit{Block} \\ \langle(\sigma, N), \mathit{Block}\rangle \to \langle(\sigma', R[\cdot]), \cdot\rangle\end{array}}{\langle(\sigma, N), \mathit{name}(v_1, v_2, ...)\rangle \to \langle(\sigma', E), \cdot\rangle}$$

4.1.2 External function calls

In this section, we will explain the rules used to evaluate external function calls. In this case, the rules written below are very similar to the ones in Section 2.2.2. For this reason, we will not give as many details.

The first two rules take the "e-call" statement and evaluate each of the arguments provided, while the rest evaluate the function itself.

$$\textbf{EVAL-EXT-FUN-CALL}\ \frac{\begin{array}{c} id \in \mathcal{C} \\ \mathbb{A}(id) = address, \qquad \Delta(address) = \sigma' \\ name \in \sigma'_{functions} \\ \text{Eval}(\sigma, Exp_{eth}) \to \langle(\sigma_1, N), v_{eth}\rangle \\ \text{Eval}(\sigma_1, Exp_{gas}) \to \langle(\sigma_2, N), v_{gas}\rangle \\ \text{Eval}(\sigma_2, Exp_1) \to \langle(\sigma_3, N), v_1\rangle \\ \text{Eval}(\sigma_3, Exp_2) \to \langle(\sigma_4, N), v_2\rangle \\ \vdots \\ \text{Eval}(\sigma_{n+1}, Exp_n) \to \langle(\sigma_{n+2}, N), v_n\rangle \\ \text{Eval}(\sigma_{n+2}, id.name.\text{value}(v_{eth}).\text{gas}(v_{gas})(v_1, v_2, \ldots, v_n)) \to \langle(\sigma_{n+2}, N), v\rangle \end{array}}{\text{Eval}(\sigma, \text{e-call } id.name.\text{value}(Exp_{eth}).\text{gas}(Exp_{gas})(Exp_1, Exp_2, \ldots, Exp_n)) \to \langle(\sigma_{n+2}, N), v\rangle}$$

If we are calling a function that is not defined in the contract *id*, then the fallback function will be executed, and since that function does not return any value, the evaluation will raise an exception.

$$\textbf{EVAL-EXT-FUN-CALL-EXC}\ \frac{\begin{array}{c} id \in \mathcal{C} \\ \mathbb{A}(id) = address, \qquad \Delta(address) = \sigma' \\ name \notin \sigma'_{functions} \\ \sigma'_{fallback} = fname \\ \text{Eval}(\sigma, Exp_{eth}) \to \langle(\sigma_1, N), v_{eth}\rangle \\ \text{Eval}(\sigma_1, Exp_{gas}) \to \langle(\sigma_2, N), v_{gas}\rangle \\ \text{Eval}(\sigma_2, id.fname.\text{value}(v_{eth}).\text{gas}(v_{gas})()) \to \langle(\sigma_2, E), \cdot\rangle \end{array}}{\text{Eval}(\sigma, \text{e-call } id.name.\text{value}(Exp_{eth}).\text{gas}(Exp_{gas})(Exp_1, Exp_2, \ldots, Exp_n)) \to \langle(\sigma_2, E), \cdot\rangle}$$

The following rules should be easy to understand, as they are almost the same rules that we saw in Section 2.2.2. The only difference is that now we need the function to return a value, which will be returned by the evaluation.

$$\textbf{EVAL-EXT-FUN}\ \frac{\begin{array}{c} \mathbb{A}(id) = address, \qquad \Delta(address) = \sigma' \\ \Omega'_\sigma = \Omega_\sigma.\text{push}(\sigma) \\ \Gamma_p\, name = Block, \qquad \Gamma_t\, name = ((p_1, \text{T}_1), (p_2, \text{T}_2), \ldots, (p_n, \text{T}_n)) \\ \langle(\sigma', N), \text{T}_1\,\text{M}\,p_1 = v_1;\ \text{T}_2\,\text{M}\,p_2 = v_2; \ldots;\ \text{T}_n\,\text{M}\,p_n = v_n; Block\rangle \to \langle(\sigma'', R[v]), \cdot\rangle \\ \Omega'_\sigma = \Omega_\sigma.\text{pop}() \end{array}}{\text{Eval}(\sigma, id.name.\text{value}(v_{eth}).\text{gas}(v_{gas})(v_1, v_2, \ldots, v_n)) \to \langle(\sigma, N), v\rangle}$$

$$\textbf{EVAL-EXT-FUN-EXC}_1 \quad \frac{\begin{array}{c}\mathbb{A}(id) = address, \qquad \Delta(address) = \sigma' \\ \Omega'_\sigma = \Omega_\sigma.\text{push}(\sigma) \\ \Gamma_p\, name = Block, \qquad \Gamma_t\, name = ((p_1, \mathrm{T}_1), (p_2, \mathrm{T}_2), \ldots, (p_n, \mathrm{T}_n)) \\ \langle (\sigma', N), \mathrm{T}_1\,\mathrm{M}\,p_1 = v_1;\ \mathrm{T}_2\,\mathrm{M}\,p_2 = v_2; \ldots;\ \mathrm{T}_n\,\mathrm{M}\,p_n = v_n; Block\rangle \to \langle (\sigma'', R[\cdot]), \cdot\rangle \\ \Omega'_\sigma = \Omega_\sigma.\text{pop}()\end{array}}{\text{Eval}(\sigma, id.name.\text{value}(v_{eth}).\text{gas}(v_{gas})(v_1, v_2, \ldots, v_n)) \to \langle (\sigma, E), \cdot\rangle}$$

$$\textbf{EVAL-EXT-FUN-EXC}_2 \quad \frac{\begin{array}{c}\mathbb{A}(id) = address, \qquad \Delta(address) = \sigma' \\ \Omega'_\sigma = \Omega_\sigma.\text{push}(\sigma) \\ \Gamma_p\, name = Block, \qquad \Gamma_t\, name = ((p_1, \mathrm{T}_1), (p_2, \mathrm{T}_2), \ldots, (p_n, \mathrm{T}_n)) \\ \langle (\sigma', N), \mathrm{T}_1\,\mathrm{M}\,p_1 = v_1;\ \mathrm{T}_2\,\mathrm{M}\,p_2 = v_2; \ldots;\ \mathrm{T}_n\,\mathrm{M}\,p_n = v_n; Block\rangle \to \langle (\sigma'', E), \cdot\rangle \\ \Omega'_\sigma = \Omega_\sigma.\text{pop}()\end{array}}{\text{Eval}(\sigma, id.name.\text{value}(v_{eth}).\text{gas}(v_{gas})(v_1, v_2, \ldots, v_n)) \to \langle (\sigma, E), \cdot\rangle}$$

4.2 Evaluating literals

To evaluate literals we provide the following rules:

$$\textbf{EVAL-BIN-OP} \quad \frac{\begin{array}{c}\text{Eval}(\sigma, Exp_1) \to \langle (\sigma', N), v_1\rangle \\ \text{Eval}(\sigma', Exp_2) \to \langle (\sigma'', N), v_2\rangle\end{array}}{\text{Eval}(\sigma, Exp_1\ \text{op}\ Exp_2) \to \langle (\sigma'', x), v_1\ \text{Trad(op)}\ v_2\rangle}$$

$$\textbf{EVAL-BIN-OP-EXC} \quad \frac{\begin{array}{c}\text{Eval}(\sigma, Exp_1) \to \langle (\sigma', x_1), v_1\rangle \\ \text{Eval}(\sigma', Exp_2) \to \langle (\sigma'', x_2), v_2\rangle \\ x_1 = E\ \vee x_2 = E\end{array}}{\text{Eval}(\sigma, Exp_1\ \text{op}\ Exp_2) \to \langle (\sigma'', E), \cdot\rangle}$$

We have used "op" to denote one of the following operators: ==, !=, >, <, >=, <=, &&, || , !, +, -, *, ** or /. We assume that both expressions have the same type. Furthermore, we assume that the operator used is compatible with their type —Boolean operators used for Booleans, integer operators used for integers, and so on—.

Evaluating constants such as numbers is as simple as translating them from their syntactic form to their semantic form.

$$\textbf{EVAL-CONST} \quad \frac{\text{Trad(const)} = v}{\text{Eval}(\sigma, \text{const}) \to \langle (\sigma, N), v\rangle}$$

The following rules are used to evaluate more complex literals, therefore Eval has to be called recursively.

$$\textbf{EVAL-ARRAY}\ \frac{\begin{array}{c}\text{Eval}(\sigma, Exp_1) \to \langle(\sigma_1, N), v_1\rangle \\ \text{Eval}(\sigma_1, Exp_2) \to \langle(\sigma_2, N), v_2\rangle \\ \vdots \\ \text{Eval}(\sigma_{n-1}, Exp_n) \to \langle(\sigma_n, N), v_n\rangle\end{array}}{\text{Eval}(\sigma, [Exp_1, Exp_2, \ldots, Exp_n]) \to \langle(\sigma_n, N), [v_1, v_2, \ldots, v_n]\rangle}$$

$$\textbf{EVAL-STRUCT}\ \frac{\begin{array}{c}\text{Eval}(\sigma, Exp_1) \to \langle(\sigma_1, N), v_1\rangle \\ \text{Eval}(\sigma_1, Exp_2) \to \langle(\sigma_2, N), v_2\rangle \\ \vdots \\ \text{Eval}(\sigma_{n-1}, Exp_n) \to \langle(\sigma_n, N), v_n\rangle\end{array}}{\text{Eval}(\sigma, (Exp_1, Exp_2, \ldots, Exp_n)) \to \langle(\sigma_n, N), [v_1, v_2, \ldots, v_n]\rangle}$$

$$\textbf{EVAL-MAP}\ \frac{\begin{array}{c}\text{Eval}(\sigma, Exp_{c_1}) \to \langle(\sigma_1, N), c_1\rangle \\ \text{Eval}(\sigma_1, Exp_{v_1}) \to \langle(\sigma_2, N), v_1\rangle \\ \text{Eval}(\sigma_2, Exp_{c_2}) \to \langle(\sigma_3, N), c_2\rangle \\ \text{Eval}(\sigma_3, Exp_{v_2}) \to \langle(\sigma_4, N), v_2\rangle \\ \vdots \\ \text{Eval}(\sigma_{2n-3}, Exp_{c_n}) \to \langle(\sigma_{2n-2}, N), c_n\rangle \\ \text{Eval}(\sigma_{2n-1}, Exp_{v_n}) \to \langle(\sigma_{2n}, N), v_n\rangle\end{array}}{\text{Eval}(\sigma, [\{Exp_{c_1} : Exp_{v_1}\}, \{Exp_{c_2} : Exp_{v_2}\}, \ldots, \{Exp_{c_n} : Exp_{v_n}\}]) \to \langle(\sigma_{2n}, N), [\{c_1 : v_1\}, \{c_2 : v_2\}, \ldots, \{c_n : v_n\}]\rangle}$$

4.3 Evaluating array functions

Each array has a function called *size* that returns the actual array size. If the array being addressed is static, this rule is quite straight-forward.

$$\textbf{EVAL-SIZE-SARRAY}\ \frac{\mathcal{T}(Exp) = \text{T}[n]}{\text{Eval}(\sigma, Exp.\text{size}()) \to \langle(\sigma, N), n\rangle}$$

If the array is not static but dynamic, it is a bit more difficult. We will not give a rule for arrays stored in memory, as it would require to define more auxiliary functions regarding the memory. These functions would be so abstract that it would not give any further information to the reader.

$$\textbf{EVAL-SIZE-DARRAY}\ \frac{\begin{array}{c}\mathcal{T}(Exp) = \text{T}[] \\ \mathcal{A}(Exp) = S \\ D(\sigma, Exp) \to (\sigma', addr) \\ [addr]_{S'_\sigma}^{\text{Size(address)}} = n\end{array}}{\text{Eval}(\sigma, Exp.\text{size}()) \to \langle(\sigma, N), n\rangle}$$

4.4 Evaluating expressions

When evaluating a generic expression, we first need to obtain its address with $\mathcal{D}$, and then we get its value with $\mathcal{G}$, which we have studied in Section 3.2. This rule can be used when we are accessing an element of a complex type stored in storage.

$$\textbf{EVAL-REST-STR}\ \frac{\begin{array}{c}\mathcal{A}(Exp) = S \\ \mathcal{D}(\sigma, Exp) \to (\sigma', addr) \\ \mathcal{G}(\sigma', (addr, \mathcal{T}(Exp))) \to v\end{array}}{\text{Eval}(\sigma, Exp) \to \langle(\sigma', N), v\rangle}$$

In case we are evaluating an expression that accesses a variable stored in memory, this is the rule that will be applied.

$$\textbf{EVAL-REST-MEM}\ \frac{\begin{array}{c}\mathcal{A}(Exp) = M \\ retrieve\_mem(\sigma, Exp) \to \langle N, v\rangle\end{array}}{\text{Eval}(\sigma, Exp) \to \langle(\sigma, N), v\rangle}$$

4.5 Address of an expression

When working with expressions stored in storage, sometimes we have used the phrase "address of an expression." If the expression is just an identifier, this is not hard to understand. In contrast, if the expression is more complex, what we mean by "its address" is the address of the element that is being accessed in the expression. For example, if "a" is an array, the address of "a[2]" would mean the address in which the second element of "a" is stored.

When we need to know the address of an identifier, we only need to return the value N_{Ψ_σ} returns.

$$\textbf{ADD-ID}\ \frac{id \in \mathbb{ID}_{\Psi_\sigma}}{\mathcal{D}(\sigma, id) \to (\sigma, N_{\Psi_\sigma}\ id)}$$

If the expression we want to know the address of is an element of an array, we will use the following two rules, depending on whether it is a dynamic or a static array.

$$\textbf{ADD-ARRAY-ST}\ \frac{\begin{array}{c}\mathcal{T}(Exp) = \text{T}[n], \quad \mathcal{T}(Exp_i) = int^* \ \vee\ \mathcal{T}(Exp_i) = \_ \\ \text{Eval}(\sigma, Exp_i) \to \langle(\sigma', N), v\rangle \\ 0 \leq v{<}n \\ \mathcal{D}(\sigma', Exp) \to (\sigma'', addr)\end{array}}{\mathcal{D}(\sigma, Exp[Exp_i]) \to (\sigma'', shift(addr, v, \text{T}))}$$

$$\textbf{ADD-ARRAY-DIN}\ \frac{\begin{array}{c}\mathcal{T}(Exp) = \mathrm{T}[], \quad \mathcal{T}(Exp_i) = int^* \ \vee\ \mathcal{T}(Exp_i) = \_ \\ \mathrm{Eval}(\sigma, Exp_i) \to \langle(\sigma', N), v\rangle \\ \mathcal{D}(\sigma', Exp) \to (\sigma'', addr) \\ [addr]_{\mathcal{A}(Exp)}^{\mathrm{Size}(\mathcal{T}(Exp))} = n \\ 0 \leq v < n\end{array}}{\mathcal{D}(\sigma, Exp[Exp_i]) \to (\sigma'', shift(\mathbf{HASH}(addr), v, \mathrm{T}))}$$

Note that the type of the expression Exp_i can either be an int (*uint8* to *uint256* or *int8* to *int256*), which we denoted by int^*, or a literal.

The rule used to find the address of an element in a mapping is very similar to the rules used for arrays.

$$\textbf{ADD-MAPPING}\ \frac{\begin{array}{c}\mathcal{T}(Exp) = \mathrm{mapping(K => T)} \\ \mathcal{T}(Exp_i) = K \ \vee\ \mathcal{T}(Exp_i) = \_ \\ \mathrm{Eval}(\sigma, Exp_i) \to \langle(\sigma', N), v\rangle \\ \mathcal{D}(\sigma', Exp) \to (\sigma'', addr)\end{array}}{\mathcal{D}(\sigma, Exp[Exp_i]) \to (\sigma'', \lceil \mathbf{HASH}(addr \cdot v)\rceil^{(l,\mathrm{T})})}$$

If we are accessing a struct, instead of using the auxiliary function *shift*, we will use $\mathcal{O}$ to find the address of the member that is being accessed. We will study this function in Section 5.5.

$$\textbf{ADD-STRUCT}\ \frac{\begin{array}{c}\mathcal{T}(Exp) = \mathrm{T}, \qquad \mathrm{T} \in \mathbb{S} \\ \mathcal{S}(\mathrm{T}) = \langle(id_1, \mathrm{T}_1), (id_2, \mathrm{T}_2), \ldots, (id_n, \mathrm{T}_n)\rangle \\ id = id_j, \quad 1 \leq j \leq n \\ \mathcal{D}(\sigma, Exp) \to (\sigma', addr) \\ \mathcal{O}(addr, id, \mathrm{T}) = addr_2\end{array}}{\mathcal{D}(\sigma, Exp.id) \to (\sigma', addr_2)}$$

5 Auxiliary functions

After explaining the main rules and semantics for the Solidity subset we chose to work with, we will now give some further details on certain functions that we have needed to define in order to give a formalization.

5.1 Typing expressions

Apart from the auxiliary function τ, we need a more complex function to infer the type of complex expressions. In the same manner as we defined the undefined location for literals, for this function we will use _ to denote the type of a literal which is not an array

literal, a struct literal or a mapping literal, thus:

$$\mathcal{T}(cte) = \_$$
$$\mathcal{T}(Exp_1 \text{ op } Exp_2) = \_$$

$$\text{TYPE-ID}_1 \ \frac{id \in N_M, \quad \tau_M(\text{id}) = \text{T}}{\mathcal{T}(\text{id}) = \text{T}} \qquad \text{TYPE-ID}_2 \ \frac{id \in N_\Psi, \qquad id \notin N_M \qquad \tau_\Psi(\text{id}) = \text{T}}{\mathcal{T}(\text{id}) = \text{T}}$$

We assume that array literals can only contain expressions of the same type.

$$\mathcal{T}([Exp_1, Exp_2, \ldots, Exp_n]) = [n]$$
$$\mathcal{T}((Exp_1, Exp_2, \ldots, Exp_n)) = struct\_lit$$
$$\mathcal{T}([\{Exp_{c_1} : Exp_{v_1}\}, \{Exp_{c_2} : Exp_{v_2}\}, \ldots, \{Exp_{c_n} : Exp_{v_n}\}]) = map\_lit$$

The type of an element of an array will be the type of the array itself.

$$\textbf{TYPE-ARRAY} \ \frac{\mathcal{T}(Exp) = \text{T}[\_] \qquad \mathcal{T}(Exp_i) = \text{int} \ \vee \ \mathcal{T}(Exp_i) = \_}{\mathcal{T}(Exp[Exp_i]) = \text{T}}$$

In contrast, the type of a struct member will be the type associated with the id.

$$\textbf{TYPE-STRUCT} \ \frac{\mathcal{T}(Exp) = \text{T}, \qquad \text{T} \in \mathbb{S} \qquad \mathcal{S}(\text{T}) = \langle (id_1, \text{T}_1), (id_2, \text{T}_2), \ldots, (id_n, \text{T}_n) \rangle \qquad id = id_j, \quad 1 \leq j \leq n}{\mathcal{T}(Exp.id) = \text{T}_j}$$

Lastly, given a mapping taking the type K as keys and T as values, each element will be of type T.

$$\textbf{TYPE-MAPPING} \ \frac{\mathcal{T}(Exp) = \text{mapping (K =>T)} \qquad \mathcal{T}(Exp_i) = \text{K} \ \vee \ \mathcal{T}(Exp_i) = \_}{\mathcal{T}(Exp[Exp_i]) = \text{T}}$$

5.2 Distinguishing value-type and reference-type variables and expressions

This auxiliary function takes either an expression or a type as input, and returns whether they are a value, a reference type, or none.

$$\begin{aligned}
\mathbb{T}(\text{bool}) = \mathbb{T}(\text{int}) = \mathbb{T}(\text{uint}) = \mathbb{T}(\text{address}) &= VT \\
\mathbb{T}(\text{T}[]) = \mathbb{T}(\text{T}[n]) = \mathbb{T}(\text{mapping (K =>T)}) &= RT \\
\mathbb{T}(cte) &= _ \\
\mathbb{T}(Exp\ op\ Exp) &= _ \\
\mathbb{T}(id) &= \mathbb{T}(\tau(id)) \\
\mathbb{T}([Exp_1, Exp_2, \ldots, Exp_n]) &= _ \\
\mathbb{T}((Exp_1, Exp_2, \ldots, Exp_n)) &= _ \\
\mathbb{T}([\{Exp_{c_1} : Exp_{v_1}\}, \{Exp_{c_2} : Exp_{v_2}\}, \ldots, \{Exp_{c_n} : Exp_{v_n}\}]) &= _
\end{aligned}$$

Every type declared as a struct is a reference type.

$$\frac{\text{T} \in \mathbb{S}}{\mathbb{T}(\text{T}) = RT}$$

For more complex expressions, we will always return RT, as we are interested in knowing about the variable being accessed.

$$\frac{\mathcal{T}(Exp) = \text{T}[\_] \qquad \mathcal{T}(Exp_i) = \text{int} \ \vee \ \mathcal{T}(Exp_i) = \_}{\mathbb{T}(Exp[Exp_i]) = RT}$$

$$\frac{\mathcal{T}(Exp) = \text{mapping(K =>T)} \qquad \mathcal{T}(Exp_i) = \text{K} \ \vee \ \mathcal{T}(Exp_i) = \_}{\mathbb{T}(Exp[Exp_i]) = RT}$$

$$\frac{\mathcal{T}(Exp) = \text{T}, \qquad \text{T} \in \mathbb{S} \qquad \mathcal{S}(\text{T}) = \langle (id_1, \text{T}_1), (id_2, \text{T}_2), \ldots, (id_n, \text{T}_n) \rangle \qquad id = id_j, \quad 1 \leq j \leq n}{\mathbb{T}(Exp.id) = RT}$$

5.3 Distinguishing state and local variables

This auxiliary function is used to check whether an expression is a state variable, a local variable, or a literal.

If the expression is a literal, this function will return _.

$$\begin{aligned}
\mathcal{L}(cte) &= _ \\
\mathcal{L}(Exp\ op\ Exp) &= _ \\
\mathcal{L}([Exp_1, Exp_2, \ldots, Exp_n]) &= _ \\
\mathcal{L}((Exp_1, Exp_2, \ldots, Exp_n)) &= _ \\
\mathcal{L}([\{Exp_{c_1} : Exp_{v_1}\}, \{Exp_{c_2} : Exp_{v_2}\}, \ldots, \{Exp_{c_n} : Exp_{v_n}\}]) &= _
\end{aligned}$$

To check whether an identifier corresponds to a state variable or not, the only thing that has to be done is checking if the id is an element of the set of all the state variables $\mathcal{ST}$.

$$\frac{id \in \mathcal{ST}}{\mathcal{L}(id) = ST}$$

$$\frac{id \notin \mathcal{ST}}{\mathcal{L}(id) = L}$$

For more complex expressions, the only thing we need to do is to check the location of the variable which is being accessed in the expression.

$$\frac{\begin{array}{c}\mathcal{T}(Exp) = \mathrm{T}[\_] \\ \mathcal{T}(Exp_i) = \mathrm{int} \ \vee \ \mathcal{T}(Exp_i) = \_ \\ \mathcal{L}(Exp) = v\end{array}}{\mathcal{L}(Exp[Exp_i]) = v}$$

$$\frac{\begin{array}{c}\mathcal{T}(Exp) = \mathrm{mapping(K => T)} \\ \mathcal{T}(Exp_i) = \mathrm{K} \ \vee \ \mathcal{T}(Exp_i) = \_ \\ \mathcal{L}(Exp) = v\end{array}}{\mathcal{L}(Exp[Exp_i]) = v}$$

$$\frac{\begin{array}{c}\mathcal{T}(Exp) = \mathrm{T}, \qquad \mathrm{T} \in \mathbb{S} \\ \mathcal{S}(\mathrm{T}) = \langle (id_1, \mathrm{T}_1), (id_2, \mathrm{T}_2), \ldots, (id_n, \mathrm{T}_n) \rangle \\ id = id_j, \quad 1 \leq j \leq n \\ \mathcal{L}(Exp) = v\end{array}}{\mathcal{L}(Exp.id) = v}$$

5.4 Checking the location of a variable

The rules that will be introduced in this section are very similar to the ones we showed in Section 5.3.

To distinguish between variables located in memory, storage or literals, we will do almost the same thing we did in the previous section.

$$\begin{aligned}
\mathcal{A}(cte) &= _ \\
\mathcal{A}(Exp\ op\ Exp) &= _ \\
\mathcal{A}([Exp_1, Exp_2, \ldots, Exp_n]) &= _ \\
\mathcal{A}((Exp_1, Exp_2, \ldots, Exp_n)) &= _ \\
\mathcal{A}([\{Exp_{c_1} : Exp_{v_1}\}, \{Exp_{c_2} : Exp_{v_2}\}, \ldots, \{Exp_{c_n} : Exp_{v_n}\}]) &= _
\end{aligned}$$

When accessing a variable declared both in memory and storage, we will give preference to the one in memory.

$$\frac{id \in N_M}{\mathcal{A}(id) = M} \qquad \frac{id \in N_\Psi, \qquad id \notin N_M}{\mathcal{A}(id) = S}$$

$$\frac{\begin{array}{c}\mathcal{T}(Exp) = \mathrm{T}[\_] \\ \mathcal{T}(Exp_i) = \mathrm{int} \ \vee \ \mathcal{T}(Exp_i) = \_ \\ \mathcal{A}(Exp) = v\end{array}}{\mathcal{A}(Exp[Exp_i]) = v}$$

$$\frac{\begin{array}{c}\mathcal{T}(Exp) = \mathrm{mapping(K =>T)} \\ \mathcal{T}(Exp_i) = \mathrm{K} \ \vee \ \mathcal{T}(Exp_i) = \_ \\ \mathcal{A}(Exp) = v\end{array}}{\mathcal{A}(Exp[Exp_i]) = v}$$

$$\frac{\begin{array}{c}\mathcal{T}(Exp) = \mathrm{T}, \qquad \mathrm{T} \in \mathbb{S} \\ \mathcal{S}(\mathrm{T}) = \langle (id_1, \mathrm{T}_1), (id_2, \mathrm{T}_2), \ldots, (id_n, \mathrm{T}_n) \rangle \\ id = id_j, \quad 1 \leq j \leq n \\ \mathcal{A}(Exp) = v\end{array}}{\mathcal{A}(Exp.id) = v}$$

5.5 Offset

This auxiliary function is very similar to the function *shift* presented in Section 3.1. In this case, instead of shifting an amount of elements in an array, $\mathcal{O}$ gets the id of the struct member we need to find the offset in order to access it. It also receives the address where the struct is located, *addr*, and the type of the struct, T.

$$\textbf{OFFSET}\ \frac{\begin{array}{c} T \in \mathbb{S} \\ \mathcal{S}(\mathrm{T}) = \langle (id_1, \mathrm{T}_1), (id_2, \mathrm{T}_2), \ldots, (id_n, \mathrm{T}_n) \rangle \\ id = id_j, \qquad 1 \leq j \leq n \\ \mathrm{Size}(\mathrm{T}_1) = v_1 \\ \mathrm{Size}(\mathrm{T}_2) = v_2 \\ \vdots \\ \mathrm{Size}(\mathrm{T}_{j-1}) = v_{j-1} \end{array}}{\mathcal{O}(addr, id, \mathrm{T}) = \lceil \ldots \lceil addr + v_1 \rceil^{(l,\mathrm{T}_2)} + v_2 \rceil^{(l,\mathrm{T}_3)} + \cdots + v_{j-2} \rceil^{(l,\mathrm{T}_{j-1})} + v_{j-1} \rceil^{(l,\mathrm{T}_j)}}$$

5.6 Default value of a type

Finding the default value of a type is not complicated. We believe that the rules presented here are quite self-expanatory.

$$\begin{aligned} \mathrm{DefVal}(\mathrm{bool}) &= \bot \\ \mathrm{DefVal}(\mathrm{address}) = \mathrm{DefVal}(\mathrm{int}) &= \mathrm{DefVal}(\mathrm{uint}) = 0 \\ \mathrm{DefVal}(\mathrm{T}[]) &= [] \\ \mathrm{DefVal}(\mathrm{mapping\ (K\ =>T)}) &= [] \end{aligned}$$

$$\frac{\mathrm{DefVal}(\mathrm{T}) = v}{\mathrm{DefVal}(\mathrm{T}[n]) = [v, \ldots^{n)}, v]}$$

$$\frac{\begin{array}{c} \mathrm{T} \in \mathbb{S} \\ \mathcal{S}(\mathrm{T}) = \langle (id_1, \mathrm{T}_1), (id_2, \mathrm{T}_2), \ldots, (id_n, \mathrm{T}_n) \rangle \\ \mathrm{DefVal}(\mathrm{T}_1) = v_1 \\ \mathrm{DefVal}(\mathrm{T}_2) = v_2 \\ \vdots \\ \mathrm{DefVal}(\mathrm{T}_n) = v_n \end{array}}{\mathrm{DefVal}(\mathrm{T}) = [v_1, v_2, \ldots, v_n]}$$

5.7 Size

This auxiliary function does not return the size of a type but rather its size in bits when located in storage, according to the rules presented in Section 3.1.

The size of simple types is quite straight forward.

$$\begin{aligned}
\text{Size(bool)} &= 1 \\
\text{Size(int8)} &= 8 \\
\text{Size(int16)} &= 16 \\
&\vdots \\
\text{Size(int256)} &= 256 \\
\text{Size(uint8)} &= 8 \\
\text{Size(uint16)} &= 16 \\
&\vdots \\
\text{Size(uint256)} &= 256 \\
\text{Size(uint)} &= \text{Size(uint256)} \\
\text{Size(int)} &= \text{Size(int256)} \\
\text{Size(address)} &= 160
\end{aligned}$$

Even for maps and dynamic arrays, as they are not stored sequentially in storage, we will say that their size is the same as the size of an unsigned integer.

$$\begin{aligned}
\text{Size(T[])} &= \text{Size(uint)} \\
\text{Size(mapping (K =>T))} &= \text{Size(uint)}
\end{aligned}$$

Things turn out to be a bit more complicated when working with static arrays and structs, as their components are stored sequentially, and they occupy a full slot.

$$\frac{\text{Size(T)} = v}{\text{Size}(T[n]) = \lceil \dots \lceil v \rceil^{(l,\text{T})} + v \rceil^{(l,\text{T})} + \dots + v \rceil^{(l,\text{T})} + v \rceil^{(l,\text{T[n]})}}$$

$$\frac{\begin{array}{c}
\text{T} \in \mathbb{S} \\
\mathcal{S}(\text{T}) = \langle (id_1, \text{T}_1), (id_2, \text{T}_2), \dots, (id_n, \text{T}_n) \rangle \\
\text{Size}(\text{T}_1) = v_1 \\
\text{Size}(\text{T}_2) = v_2 \\
\vdots \\
\text{Size}(\text{T}_n) = v_n
\end{array}}{\text{Size}(T) = \lceil \dots \lceil v_1 \rceil^{(l,\text{T}_2)} + v_2 \rceil^{(l,\text{T}_3)} + \dots + v_{n-1} \rceil^{(l,\text{T}_n)} + v_n \rceil^{(l,\text{T})}}$$

6 Example

In this last chapter, we will go through the different rules that will be used when deploying the contract introduced in Section 1.6. The only difference between the code

presented there and the code that will be executed now is that both statements *worker(1, false, 5, addr)* and *shift(5)*, corresponding to struct literals, have been replaced by *(1, false, 5, addr)* and *(5)*, respectively, so that the code matches the semantics proposed in Section 1.5.

The first rule that will be used is INIT.

$$\textbf{INIT}\ \frac{\textbf{COMP}\ \dfrac{\begin{array}{c}\langle(\sigma,N),\text{uint company\_number} = 1;\rangle \to \langle(\sigma_1,N),\cdot\rangle \\ \langle(\sigma_1,N),\text{address boss\_identifier;}\rangle \to \langle(\sigma_2,N),\cdot\rangle \\ \langle(\sigma_2,N),\text{mapping(address =>worker) workers;}\rangle \to \langle(\sigma_3,N),\cdot\rangle \\ \langle(\sigma_3,N),\text{shift[] shifts;}\rangle \to \langle(\sigma_4,N),\cdot\rangle\end{array}}{\langle(\sigma,N),\text{uint company\_number} = 1;\ \text{address boss\_identifier;}\ \text{mapping(address =>worker) workers;}\ \text{shift[] shifts;}\rangle \to \langle(\sigma_4,N),\cdot\rangle}}{\langle\text{contract company}\{Body\},(v)\rangle \Rightarrow \langle(\sigma_4,N),\text{company}(v)\rangle}\ initialise(Body) = \sigma$$

Here, the *Body* section corresponds to the code inside the contract definition. And v is a 20-byte value. For this example, we have chosen the value

$$0xdCad3a6d3569DF655070DEd06cb7A1b2Ccd1D3AF,$$

although we will write v to simplify the notation. In order to minimise the space used, the COMP rule has been slightly modified, so that only one rule is used for the concatenation of four statements. We will do this in more occasions throughout the example.

Now we will take a look at each of the four statements appearing in the *Body.vars* section, before running the contract's constructor.

The first state variable is *company_number*, which will be initialised to 1:

$$\textbf{DEF-VAL-STR}_\textbf{1}\ \frac{\begin{array}{c}\text{Eval}(\sigma,1) \to \langle(\sigma,N),1\rangle \\ update\_state(\sigma,\text{uint},\text{company\_number}) = \sigma' \\ \mathcal{U}(\sigma',(0,\text{uint},1)) \to \sigma_1\end{array}}{\langle(\sigma,N),\text{uint company\_number} = 1;\rangle \to \langle(\sigma_1,N),\cdot\rangle}$$

The auxiliary function *update_state* modifies σ to σ' in the following manner:

$$\sigma' : \begin{cases}\tau_{\Psi'}\ \text{company\_number} = \text{uint} \\ N_{\Psi'}\ \text{company\_number} = \lceil 0 \rceil^{(256,\text{uint})} = 0 \\ \Lambda' = \Lambda \uparrow (256,\text{uint}) = 256\end{cases}$$

Now σ has been updated so that σ' contains both the type and the address assigned to *company_number*, and the pointer Λ contains the first empty address in the storage.

To finish with this first declaration, we have to take a look at the rule UPDATE-VALUE, used to store the value that corresponds to *company_number* in its address.

$$\textbf{UPDATE-VALUE}\ \frac{[0]^{256}_{S_{\sigma'}} = 1}{\mathcal{U}(\sigma', (0, \text{uint}, 1)) \to \sigma_1}$$

As we have seen in Section 3.1, $\mathcal{U}$ takes a state and a tuple containing the address, the type of the value to be stored, and the value itself, and stores in storage, updating the state.

For the next three declarations, there is no initialisation, so they will take the default value for their type. The rules used for the second declaration are very similar to the ones used to declare *company_number*:

$$\textbf{DEF-VAL-STR}_\textbf{2}\ \frac{\begin{array}{c}\text{DefVal(address)} = 0 \\ \textit{update\_state}(\sigma_1, \text{address}, \text{boss\_identifier}) = \sigma_1' \\ \mathcal{U}(\sigma_1', (256, \text{address}, 0)) \to \sigma_2\end{array}}{\langle(\sigma_1, N), \text{address boss\_identifier;}\rangle \to \langle(\sigma_2, N), \cdot\rangle}$$

The updated state will have the following modifications:

$$\sigma_1' : \begin{cases} \tau_{\Psi_1'}\ \text{boss\_identifier} = \text{address} \\ N_{\Psi_1'}\ \text{boss\_identifier} = \lceil 256 \rceil^{(256,\text{address})} = 256 \\ \Lambda_1' = \Lambda_1 \uparrow (256, \text{address}) = 416 \end{cases}$$

$$\textbf{UPDATE-VALUE}\ \frac{[256]^{256}_{S_{\sigma_1'}} = 0}{\mathcal{U}(\sigma_1', (256, \text{address}, 0)) \to \sigma_2}$$

The last two state variable declarations are slightly different, due to the fact that their type is not simple, but rather they are a mapping and an array, respectively.

$$\textbf{DEF-VAL-STR}_\textbf{2}\ \frac{\begin{array}{c}\text{DefVal(mapping(address =>worker))} = [] \\ \textit{update\_state}(\sigma_2, \text{mapping(address =>worker)}, \text{workers}) = \sigma_3 \\ \mathcal{U}(\sigma_3, (512, \text{mapping(address =>worker)}, [])) \to \sigma_3\end{array}}{\langle(\sigma_2, N), \text{mapping(address =>worker) workers;}\rangle \to \langle(\sigma_3, N), \cdot\rangle}$$

The state σ_2 is updated like we did before. In this case, note how the variable is not being saved from the first free address, 416, but rather from 512, which corresponds to the first free address aligned to the size of the blocks. This is because of the Solidity rules presented at the start of Chapter 3.

$$\sigma_3 : \begin{cases} \tau_{\Psi_3}\ \text{workers} = \text{mapping(address => worker)} \\ N_{\Psi_3}\ \text{workers} = \lceil 416 \rceil^{(256,\text{mapping(address => worker))}} = 512 \\ \Lambda_3 = \Lambda_2 \uparrow (256, \text{mapping(address => worker)}) = 768 \end{cases}$$

Now, when saving the value of the mapping, notice that no action is taken; the default value for the mapping is an empty map, and no value is stored in the address that corresponds to it. Therefore, the state will remain intact.

$$\textbf{UPDATE-VALUE}\ \frac{}{\mathcal{U}(\sigma_3, (512, \text{mapping(address =>worker)}, [])) \to \sigma_3}$$

Finally, the last declaration is fairly similar to the ones we have already seen.

$$\textbf{DEF-VAL-STR}_\textbf{2}\ \frac{\begin{array}{c}\text{DefVal(shift[])} = [] \\ \mathit{update\_state}(\sigma_3, \text{shift[]}, \text{shifts}) = \sigma_3' \\ \mathcal{U}(\sigma_3', (768, \text{shift[]}, [])) \to \sigma_4\end{array}}{\langle(\sigma_3, N), \text{shift[] shifts;}\rangle \to \langle(\sigma_4, N), \cdot\rangle}$$

$$\sigma_3' : \begin{cases} \tau_{\Psi_3'}\ \text{shifts} = \text{shift[]} \\ N_{\Psi_3'}\ \text{shifts} = \lceil 768 \rceil^{(256,\text{shift[]})} = 768 \\ \Lambda_3' = \Lambda_3 \uparrow (256, \text{shift[]}) = 1024 \end{cases}$$

In this case, the value stored in the address assigned to the array is its size, so 0 will be stored.

$$\textbf{UPDATE-VALUE}\ \frac{[768]^{256}_{S_{\sigma_3'}} = 0}{\mathcal{U}(\sigma_3', (768, \text{shift[]}, [])) \to \sigma_4}$$

After all the declarations, the construction is executed. We show the following rule, TRANSITION+COMP, which is a mixture of both rules so that the notation is simpler, and easier for the reader to understand. When the constructor is executed, first the arguments are declared in the memory, and given their initial value, and then the code assigned to the constructor is run.

$$\textbf{TRANSITION+COMP}\ \frac{\begin{array}{c}\langle(\sigma_4, N), \text{address } M \text{ a} = v;\rangle \to \langle(\sigma_5, N), \cdot\rangle \\ \langle(\sigma_5, N), \text{boss\_identifier} = \text{a;}\rangle \to \langle(\sigma_6, N), \cdot\rangle \\ \langle(\sigma_6, N), \text{worker } S \text{ boss} = (1, \text{false, 5, boss\_identifier});\rangle \to \langle(\sigma_7, N), \cdot\rangle \\ \langle(\sigma_7, N), \text{workers[boss\_identifier]} = \text{boss;}\rangle \to \langle(\sigma_8, N), \cdot\rangle \\ \langle(\sigma_8, N), \text{shifts.push(shift(5));}\rangle \to \langle(\sigma_9, N), \cdot\rangle \\ \langle(\sigma_9, N), \text{bool } M \text{ b} = \text{e-call comp\_areas(boss\_identifier, 1);}\rangle \to \langle(\sigma_{10}, N), \cdot\rangle \\ \langle(\sigma_{10}, N), \text{return;}\rangle \to \langle(\sigma_{10}, R[\cdot]), \cdot\rangle\end{array}}{\langle(\sigma_4, N), \text{company}(v)\rangle \to \langle(\sigma_{10}, N), \cdot\rangle}$$

Note how a *return* statement has been added to the original code, so that the semantics proposed work properly, with no need to give extra definitions to treat the constructor function any different.

In the following pages we will be looking at each of the instructions that are being executed.

The first instruction corresponds to the declaration of the local variable a, which is initialised to v. The evaluation is straightforward, as v is a constant.

$$\textbf{MEM}\ \frac{\begin{array}{c}\text{Eval}(\sigma_4, v) \to \langle(\sigma_4, N), v\rangle \\ \mathit{emplace\_mem}(\sigma_4, a, \text{address}, v) = \sigma_5\end{array}}{\langle(\sigma_4, N), \text{address } M \text{ a} = v;\rangle \to \langle(\sigma_5, N), \cdot\rangle}$$

The second instruction is an assignment. The variable *boss_identifier*, which is a state variable located in storage is given the value of a, located in memory. Therefore we will use ASS-STR-MEM. The value of a will be copied into *boss_identifier*'s address. First, we retrieve the value from memory, then we find out where *boss_identifier* is located in storage using $\mathcal{D}$, studied in Section 4.5, and finally update its value, using $\mathcal{U}$.

$$\textbf{ASS-STR-MEM}\ \frac{\begin{array}{c}\mathit{retrieve\_mem}(\sigma_5, \text{a}) \to \langle N, v\rangle \\ \mathcal{D}(\sigma_5, \text{boss\_identifier}) \to (\sigma_5, 256) \\ \mathcal{U}(\sigma_5, (256, \text{address}, v)) \to \sigma_6\end{array}}{\langle(\sigma_5, N), \text{boss\_identifier} = \text{a};\rangle \to \langle(\sigma_6, N), \cdot\rangle}$$

Because *boss_identifier* is the identifier of a variable, ADD-ID returns the output of the function N_{σ_5}, which is 256.

$$\textbf{ADD-ID}\ \frac{}{\mathcal{D}(\sigma_5, \text{boss\_identifier}) \to (\sigma_5, N_{\Psi_{\sigma_5}}\text{boss\_identifier}(= 256))}$$

The process of updating the value is very similar to the ones we have already seen, due to the fact that *address* is a simple type.

$$\textbf{UPDATE-VALUE}\ \frac{[256]^{256}_{S_{\sigma_5}} = v}{\mathcal{U}(\sigma_5, (256, \text{address}, v)) \to \sigma_6}$$

The next instruction corresponds to the declaration of a struct. As structs are complex types, the rules will be a bit more intricate.

$$\textbf{STORAGE-LIT}\ \frac{\begin{array}{c}\text{Eval}(\sigma_6, (1, \text{false}, 5, \text{boss\_identifier})) \to \langle(\sigma_6, N), [1, \bot, 5, v]\rangle \\ \mathit{update\_state}(\sigma_6, \text{worker}, \text{boss}) = \sigma_6' \\ \mathcal{U}(\sigma_6', (1024, \text{worker}, [1, \bot, 5, v])) \to \sigma_7\end{array}}{\langle(\sigma_6, N), \text{worker } S \text{ boss} = (1,\ \text{false},\ 5,\ \text{boss\_identifier});\rangle \to \langle(\sigma_7, N), \cdot\rangle}$$

To evaluate the tuple $(1, \text{false}, 5, \text{boss\_identifier})$, we will have to evaluate each one of its components. For the first three components, as they are constants, the rule EVAL-CONST is used. On the other hand, for the last rule we will have to use EVAL-REST-STR.

$$\textbf{EVAL-STRUCT}\ \frac{\begin{array}{c}\text{Eval}(\sigma_6, 1) \to \langle(\sigma_6, N), 1\rangle \\ \text{Eval}(\sigma_6, \text{false}) \to \langle(\sigma_6, N), \bot\rangle \\ \text{Eval}(\sigma_6, 5)) \to \langle(\sigma_6, N), 5\rangle \\ \text{Eval}(\sigma_6, \text{boss\_identifier}) \to \langle(\sigma_6, N), v\rangle\end{array}}{\text{Eval}(\sigma_6, (1, \text{false}, 5, \text{boss\_identifier})) \to \langle(\sigma_6, N), [1, \bot, 5, v]\rangle}$$

To evaluate *boss_identifier*, we first need to know its address, and then get the value from storage, using $\mathcal{G}$, studied in Section 3.2.

$$\textbf{EVAL-REST-STR}\ \frac{\begin{array}{c}\mathcal{D}(\sigma_6, \text{boss\_identifier}) \to (\sigma_6, 256) \\ \mathcal{G}(\sigma_6, (256, \text{address})) \to v\end{array}}{\text{Eval}(\sigma_6, \text{boss\_identifier}) \to \langle(\sigma_6, N), v\rangle}$$

The rule used for $\mathcal{D}$ is ADD-ID, which we have already seen. Now, the rule used for $\mathcal{G}$ is the following:

$$\textbf{GET-VALUE}\ \frac{[256]^{256}_{S_{\sigma_6}} = v}{\mathcal{G}(\sigma_6, (256, \text{address})) \to v}$$

After evaluating the tuple, the state is updated, in the same manner that we have already seen.

$$\sigma'_6 : \begin{cases} \tau_{\Psi'_6}\ \text{boss} = \text{worker} \\ N_{\Psi'_3}\ \text{boss} = \lceil 1024 \rceil^{(256,\text{worker})} = 1024 \\ \Lambda'_6 = \Lambda_6 \uparrow (256, \text{worker}) = 1536 \end{cases}$$

Finally, the values are stored in storage. Each value is stored sequentially starting from the address given to *boss*, and the state is updated accordingly after each value is stored.

$$\textbf{UPDATE-STRUCT}\ \frac{\begin{array}{c}\mathcal{U}(\sigma'_6, (1024, \text{uint}, 1)) \to \sigma''_6 \\ \mathcal{U}(\sigma''_6, (1280, \text{bool}, \bot)) \to \sigma'''_6 \\ \mathcal{U}(\sigma'''_6, (1281, \text{uint8}, 5)) \to \sigma''''_6 \\ \mathcal{U}(\sigma''''_6, (1289, \text{address}, v)) \to \sigma_7\end{array}}{\mathcal{U}(\sigma'_6, (1024, \text{worker}, [1, \bot, 5, v])) \to \sigma_7}$$

The next instruction corresponds to the assignment of a new element in the mapping.

$$\textbf{ASS-STR-STR}_\textbf{1}\ \frac{\begin{array}{c}\text{Eval}(\sigma_7, \text{boss}) \to \langle(\sigma_7, N), [1, \bot, 5, v]\rangle \\ \mathcal{D}(\sigma_7, \text{workers}[\text{boss\_identifier}]) \to (\sigma_7, \lceil \textbf{HASH}(512 \cdot v) \rceil^{(256,\text{worker})}) \\ \mathcal{U}(\sigma_7, (\lceil \textbf{HASH}(512 \cdot v) \rceil^{(256,\text{worker})}, \text{worker}, [1, \bot, 5, v])) \to \sigma_8\end{array}}{\langle(\sigma_7, N), \text{workers}[\text{boss\_identifier}] = \text{boss};\rangle \to \langle(\sigma_8, N), \cdot\rangle}$$

The first thing that is done is getting the value of *boss*. We will do so by using EVAL-REST-STR, ADD-ID (not shown) and GET-STRUCT. The value of *boss* is retrieved from the memory after getting its address.

$$\textbf{EVAL-REST-STR}\ \frac{\begin{array}{c}\mathcal{D}(\sigma_7, \text{boss}) \to (\sigma_7, 1024) \\ \mathcal{G}(\sigma_7, (1024, \text{worker})) \to [1, \bot, 5, v]\end{array}}{\text{Eval}(\sigma_7, \text{boss}) \to \langle(\sigma_7, N), [1, \bot, 5, v]\rangle}$$

To get the value of the struct, the value of each of its members is retrieved one at a time, and then put together.

$$\textbf{GET-STRUCT}\ \frac{\begin{array}{c}\mathcal{G}(\sigma_7, (1024, \text{uint})) \to 1 \\ \mathcal{G}(\sigma_7, (1280, \text{bool})) \to \bot \\ \mathcal{G}(\sigma_7, (1281, \text{uint8})) \to 5 \\ \mathcal{G}(\sigma_7, (1289, \text{address})) \to v\end{array}}{\mathcal{G}(\sigma_7, (1024, \text{worker})) \to [1, \bot, 5, v]}$$

After evaluating the struct, we will get the address that corresponds to the element *boss_identifier* of the mapping *workers*. To do so, the rule ADD-MAPPING is used:

$$\textbf{ADD-MAPPING}\ \frac{\begin{array}{c}\text{Eval}(\sigma_7, \text{boss\_identifier}) \to \langle(\sigma_7, N), v\rangle \\ \mathcal{D}(\sigma_7, \text{workers}) \to (\sigma_7, 512)\end{array}}{\mathcal{D}(\sigma_7, \text{workers}[\text{boss\_identifier}]) \to (\sigma_7, \lceil \textbf{HASH}(512 \cdot v)\rceil^{(256,\text{worker})})}$$

Both rules used for this rule have already been covered in the example, so there is no need to show them again.

To finish, the value is stored using $\mathcal{U}$. This rule has already been covered for the same type, in a different address, in the rule UPDATE-STRUCT appearing above.

The next rule deals with the instruction *push*, which will insert a new value in our array *arr*.

$$\textbf{PUSH}_\textbf{2}\ \frac{\begin{array}{c}\text{Eval}(\sigma_8, (5)) \to \langle(\sigma_8, N), [5]\rangle \\ \mathcal{D}(\sigma_8, \text{shifts}) \to (\sigma_8, 768) \\ [768]^{256}_{S_{\sigma_8}} = 1 \\ \mathcal{U}(\sigma_8, (\lceil \textbf{HASH}(768)\rceil^{(256,\text{shift})}, \text{shift}, [5])) \to \sigma_9\end{array}}{\langle(\sigma_8, N), \text{shifts.push}((5));\rangle \to \langle(\sigma_9, N), \cdot\rangle}$$

All the rules that appear in PUSH_2 have been covered above, and therefore we believe that there is no point in showing them again.

The following instruction is the most intricate, due to the function call statement. We will go through it in detail.

$$\textbf{MEM-OTHR}\ \frac{\begin{array}{c}\text{Eval}(\sigma_9, \text{e-call comp\_areas(addr, 1)}) \to \langle(\sigma_9', N), \bot\rangle \\ \textit{emplace\_mem}(\sigma_9', \text{b}, \text{bool}, \bot) = \sigma_{10}\end{array}}{\langle(\sigma_9, N), \text{bool M b = e-call comp\_areas(boss\_identifier, 1);}\rangle \to \langle(\sigma_{10}, N), \cdot\rangle}$$

First the function is evaluated, and then the value returned is stored in the memory.

To evaluate the e-call statement we need to use EVAL-FUN-CALL.

$$\textbf{EVAL-FUN-CALL}\ \frac{\begin{array}{c}\text{Eval}(\sigma_9, \text{boss\_identifier}) \to \langle(\sigma_9, N), v\rangle \\ \text{Eval}(\sigma_9, 1) \to \langle(\sigma_9, N), 1\rangle \\ \text{Eval}(\sigma_9, \text{comp\_areas}(v,\ 1)) \to \langle(\sigma_9', N), \bot\rangle\end{array}}{\text{Eval}(\sigma_9, \text{e-call comp\_areas(addr, 1)}) \to \langle(\sigma_9', N), \bot\rangle}$$

EVAL-FUN-CALL will evaluate each of the arguments given to the function, and then runs it. We will not cover the evaluation of the arguments as it has been done already.

The actual evaluation of the function is done with EVAL-FUN. We have decided to mix EVAL-FUN with COMP, so that it is easier to follow the derivation process.

$$\textbf{EVAL-FUN}\ \frac{\textbf{COMP}\ \dfrac{\begin{array}{c}\langle(\sigma_9, N), \text{address } M \text{ ident} = v\rangle \to \langle(\sigma_{9_1}, N), \cdot\rangle \\ \langle(\sigma_{9_1}, N), \text{uint8 } M \text{ area\_num} = 1\rangle \to \langle(\sigma_{9_2}, N), \cdot\rangle \\ \langle(\sigma_{9_2}, N), \text{return area\_num} > \text{workers[ident].area}\rangle \to \langle(\sigma_9', R[\bot]), \cdot\rangle\end{array}}{\langle(\sigma_9, N), \text{address } M \text{ ident} = v;\ \text{uint8 } M \text{ area\_num} = 1; \textit{Block}\rangle \to \langle(\sigma_9', R[\bot]), \cdot\rangle}}{\text{Eval}(\sigma_9, \text{comp\_areas}(v,\ 1)) \to \langle(\sigma_9', N), \bot\rangle}$$

The first two declarations will be done with MEM-OTHR, which has already been covered, and therefore gives no new information to the reader. On the other hand, it is interesting to show how the *return* statement is evaluated. We will show RETURN-VAL and EVAL-BIN-OP at the same time. EVAL-BIN-OP evaluates both operands and returns the result of the operation.

$$\textbf{RETURN-VAL}\ \frac{\textbf{EVAL-BIN-OP}\ \dfrac{\begin{array}{c}\text{Eval}(\sigma_{9_2}, \text{area\_num}) \to \langle(\sigma_{9_2}, N), 1\rangle \\ \text{Eval}(\sigma_{9_2}, \text{workers[ident].area}) \to \langle(\sigma_{9_2}, N), 5\rangle\end{array}}{\text{Eval}(\sigma_{9_2}, \text{area\_num} > \text{workers[ident].area}) \to \langle(\sigma_{9_2}, N), 1 > 5 (= \bot)\rangle}}{\langle(\sigma_{9_2}, N), \text{return area\_num} > \text{workers[ident].area}\rangle \to \langle(\sigma_9', R[\bot]), \cdot\rangle}$$

The first evaluation is of no interest, because it is located in memory. The rule used for it is EVAL-REST-MEM.

To evaluate *workers[ident].area* we will use EVAL-REST-STR. This rule, which has already been seen before, finds out the address of a given expression and retrieves the

value from storage. We will not cover the rule used for $\mathcal{G}$ as it is just GET-VALUE, which has already been shown.

$$\textbf{EVAL-REST-STR}\ \frac{\begin{array}{c}\mathcal{D}(\sigma_{9_2}, \text{workers[ident].area}) \to (\sigma_{9_2}, \lceil \mathbf{HASH}(512 \cdot v) \rceil^{(256,\text{worker})} + 257) \\ \mathcal{G}(\sigma_{9_2}, (\lceil \mathbf{HASH}(512 \cdot v) \rceil^{(256,\text{worker})} + 257, \text{uint8})) \to 5\end{array}}{\text{Eval}(\sigma_{9_2}, \text{workers[ident].area}) \to \langle (\sigma_{9_2}, N), 5 \rangle}$$

What is interesting is showing how the address of a complex expression is calculated. We will use both ADD-STRUCT and ADD-MAPPING to do this.

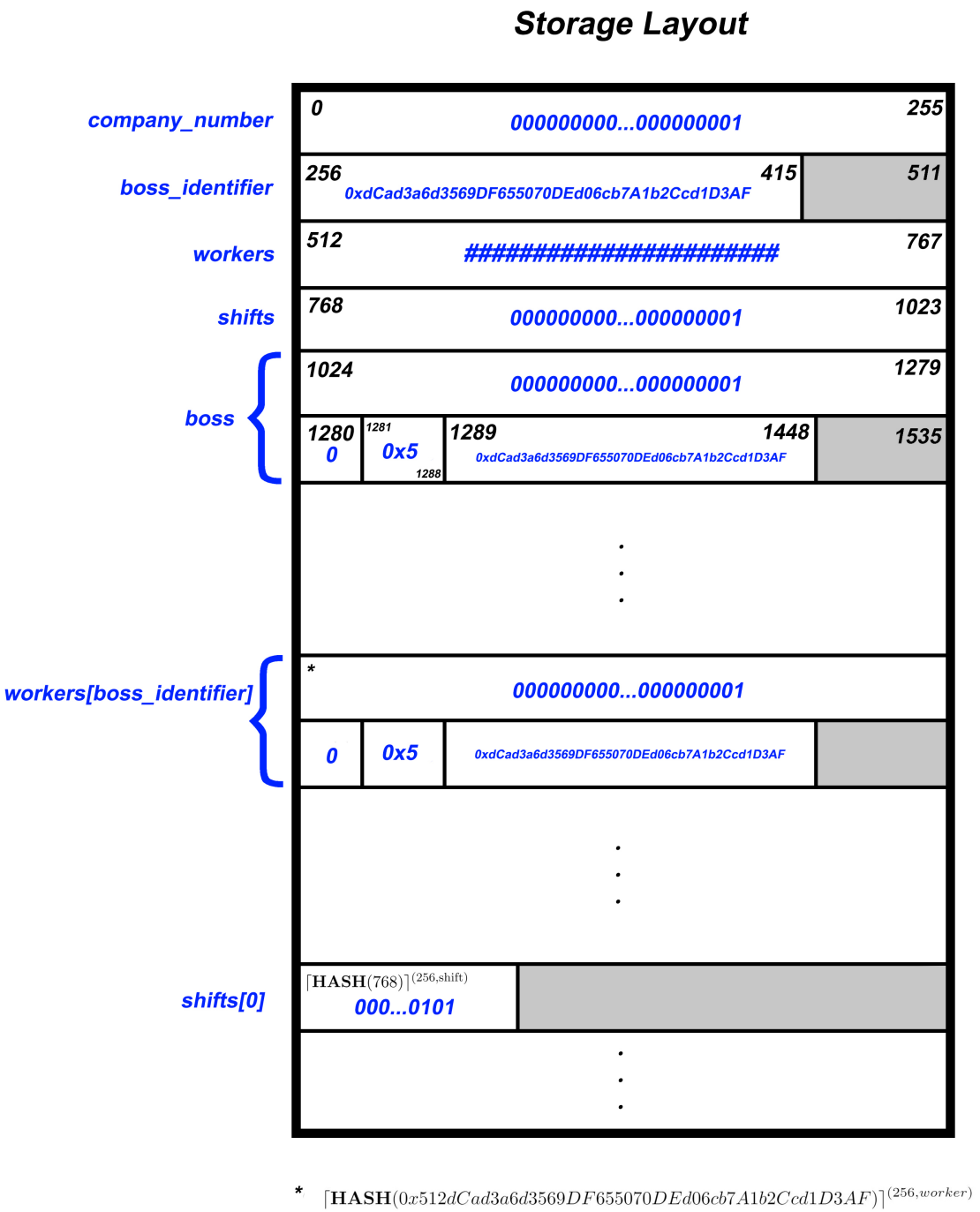
First, the address of *workers[ident]* is calculated. Once that is done, the value we are interested in is the address of *workers[ident]* plus an offset so that we access the third member.

$$\textbf{ADD-STRUCT}\ \frac{\textbf{ADD-MAPPING}\ \dfrac{\begin{array}{c}\text{Eval}(\sigma_{9_2}, \text{ident}) \to \langle (\sigma_{9_2}, N), v \rangle \\ \mathcal{D}(\sigma_{9_2}, \text{workers}) \to (\sigma_{9_2}, 512)\end{array}}{\mathcal{D}(\sigma_{9_2}, \text{workers[ident]}) \to (\sigma_{9_2}, \lceil \mathbf{HASH}(512 \cdot v) \rceil^{(256,\text{worker})})} \\ \mathcal{O}(\lceil \mathbf{HASH}(512 \cdot v) \rceil^{(256,\text{worker})}, \text{area}, \text{worker}) = \lceil \mathbf{HASH}(512 \cdot v) \rceil^{(256,\text{worker})} + 257}{\mathcal{D}(\sigma_{9_2}, \text{workers[ident].area}) \to (\sigma_{9_2}, \lceil \mathbf{HASH}(512 \cdot v) \rceil^{(l,\text{worker})} + 257)}$$

The last rule used is RETURN, which will only change the execution status to $R[\cdot]$.

$$\textbf{RETURN}\ \frac{}{\langle (\sigma_{10}, N), \text{return;} \rangle \to \langle (\sigma_{10}, R[\cdot]), \cdot \rangle}$$

To finish the example, we will show how the storage would look like after the execution of the constructor function. Note that the value in the addresses 512 to 767 is unknown. The structure of *workers[ident]* is the same as the structure of *boss*, and therefore the address numbers have not been included, for simplicity. Also, *shifts[0]* is a struct containing only a signed integer of 16 bits of size, with value 5.



* $\lceil \mathbf{HASH}(0x512dCad3a6d3569DF655070DEd06cb7A1b2Ccd1D3AF) \rceil^{(256,worker)}$

7 Implementation

To conclude this work, we have decided to give an implementation in Maude for most of the semantics presented previously. It is important to note that it is not fully executable, since there are some rules missing, but we believe that the code provided can be extended so that it will be possible to perform formal verification of contracts using it.

Because the semantics shown in [7] were executable in K, and due to the fact that our semantics follow the same structure as theirs, everything shown in the previous sections should be executable in K.

Although our first idea was to give an implementation in K, we found it rather complicated to use, and thus decided to switch to Maude. The code is available in this GitHub repository.

The implementation is divided in nine files, as we believe that this way the code is easier to understand.

The files are the following:

- Value.maude
- Types.maude
- Global.maude
- EssentialFunctions.maude
- Aux_Ops.maude
- Storage.maude
- Memory.maude
- Evaluations.maude
- Transitions.maude

We will now give a brief overview of each of them.

7.1 Value

In this file the type *Valor* is defined. *Valor* will be the sort given to values in our semantics. As we have seen throughout the previous sections, a value can either be simple, such as a number or a Boolean, but it can also be a list of simple values, a list containing lists of values, a list containing both lists and simple values, and so on.

In order to create lists of lists, we have created a wrapper in module MILIST, which takes a list, and returns a *MiList*, containing it inside square brackets.

Then, we have defined the types *SValor* and *CValor* in their respective modules. SValor is a wrapper for the types *Int* and *Bool*, and CValor is defined as a wrapper for a SValor, a MiList of Svalor, a MiList containing MiLists of Svalor, or a key-value element,

formed by an Svalor and a CValor.

Lastly, the sort Valor is defined as a List of CValor. Note that this does not accurately represent all the possible values that can be returned when evaluating an expression as the "level" of recursion here is only three, whilst in an example we could create more complex values.

In order to make this clearer, here we show some of the code in Value.maude.

```
fmod MILIST{X :: TRIV} is
   sort MiList{X} .
   op [_] : List{X} -> MiList{X} .
endfm

fmod SVALOR is
    sort SValor .
    op SV(_) : Int -> SValor .
    op SV(_) : Bool -> SValor .
endfm

fmod CVALOR is
    sort CValor .
    op CV(_) : SValor -> CValor .
    op CV(_) : MiList{MiList{SValor}} -> CValor .
    op CV(_) : MiList{SValor} -> CValor .
    op _:_ : SValor CValor -> CValor .
endfm

fmod VALOR is
    sort Valor .
    subsort List{CValor} < Valor .
endfm
```

Extract from Value.maude

7.2 Types

This section is quite simple; we have only stated the different types that we have used in this text: bool, address, int8, . . . , int256, uint8, . . . uint256, mappings, arrays, and structs.

We have four main type sorts:

- Stype, which includes all the simple types we use in our semantics. That is: booleans, integers, unsigned integers, and addresses.
- StructType, which is the type used for every struct defined in the code. Thus, each struct defined in the code can be defined in our modules like so:

  op nameofmystruct : -> StructType .

It is also a subsort of Ctype.

- Ctype, containing mappings, arrays and StructType.
- Tipo, which is used as the supertype for both Stype and Ctype.

Finally, in this module the pairs of String and Tipo are also defined, which will be used in functions like $\mathcal{S}$ or Γ_t, defined in Section 1.7.

7.3 Global

This file contains three modules: GLOBAL, SYNTAX and ESTADO.

GLOBAL is used to declare all the global variables and functions that are used in the verification, presented in Section 1.7, except for those that return code, like Γ_p and the variable $\sigma_{fallback}$. Here we define the function and event names, the outcome of Γ_t, the name of the state variables and the structs defined, alongside with the values of the function $\mathcal{S}$.

SYNTAX contains the syntax of expressions and statements, as shown in Section 1.5, alongside with the definition of the function Γ_p and the constant $\sigma_{fallback}$.

Finally, in module ESTADO there are three main definitions: the environment, the execution status and the tuple that we have called *Configuracion*, which is the outcome of both Eval and the result of an execution step. It contains a status, the execution status, and a value.

The environment is created using the operator *Env*:

$$\mathrm{Env}(N_\Psi, \tau_\Psi, \Psi, N_M, \tau_M, M, \mathbb{ID}_\Psi, \mathbb{ID}_M, \Lambda)$$

The first six members of the environment are declared as maps, both $\mathbb{ID}_\Psi$ and $\mathbb{ID}_M$ are declared as sets, and Λ is an integer.

Note that the initialisation of the global variables and functions has to be done *ad hoc* for each verification.

7.4 EssentialFunctions

This file contains only one module, with the definitions for several auxiliary functions that we have worked with along this text.

We have defined the functions Size and DefVal, introduced in Sections 5.7 and 5.6, respectively.

After those, we have defined auxiliary functions that deal with addresses, like functions $\lceil \mathrm{addr} \rceil^{(l, Type)}$ and $\mathrm{addr} \uparrow (l, Type)$, presented in Section 1.7.

We have also defined the functions $shift(addr, n, \mathrm{T})$, and $\mathcal{O}$, presented in Sections 3.1

and 5.5, respectively. In order to define $\mathcal{O}$ correctly, we have needed three auxiliary functions, *discard*, *offsetAux1* and *offsetAux2*.

The functions $\mathbf{HASH}(v)$ and $v_1 \cdot v_2$, used to hash a value and to concatenate two values, respectively, have been defined with no concrete implementation, so that the interpreter can use them, without returning an actual value.

7.5 Aux_Ops

The file Aux_Ops contains a single module in which the rest of the functions and rules presented in Section 5 are defined.

Their definitions are quite self-explanatory since the code is almost a direct translation from the semantics presented in this work to Maude.

7.6 Storage

The file Storage also contains a single module, in which we have defined the rules for functions $\mathcal{U}$ and $\mathcal{G}$, introduced in Chapter 3. And the auxiliary functions *update_state* and *update_state_pointer*, presented in Sections 2.4.1 and 2.4.2, respectively.

The main difference between the rules that appeared in Chapter 3 and the code in the file is that here we have had to use recursive functions in order to code properly the rules that made use of the "..." abstraction to work with a non-fixed number of values $v_1, v_2, \ldots, v_n$.

7.7 Memory

This is a file containing only the abstract definitions of the auxiliary functions that deal with the memory that we have used in our semantics: *retrieve_mem*, *emplace_mem*, *ref_mem*, *update_mem*, and *update_ref_mem*.

In order for the module to work properly, a model for the memory should be chosen, along with precise rules for it.

7.8 Evaluations and Transitions

In the last two files, Evaluations and Transitions, we have coded the rules appearing in Chapters 2 and 4, respectively.

We have not included either function calls, both as statements and evaluations, or events. If one wants to be able to verify a full contract using our semantics in Maude, the rules for these statements should be provided.

7.9 Short Example

In order to show the potential use of our implementation, and to show that what we already have is executable, we have decided to run a short example based on the instructions shown in the contract code in Section 1.6. The code that we will be running is the following:

```
    uint companyNumber = 1;
    address bossIdentifier;
    mapping(address => worker) workers;
    shift[] shifts;
    worker S boss = worker(1, false, 5, bossIdentifier);
    workers[bossIdentifier] = boss;
```

Code that is going to be executed

Note that we assume that the structs *worker* and *shift* have been correctly defined and processed by the interpreter. Also, the code that we will be running is a mixture of state and local variable declarations; in this case we want to show how our modules work, and therefore we are not concerned about the correction of the code; in an actual contract code, state and local variable declarations do not appear together.

We will run each of the instructions separately, starting with an empty environment, and taking the resulting environment as the input for the next instruction, so that each step is clearer to the reader.

In our opinion, the image shown at the end of Chapter 6 will help the reader understand how variables are stored in storage in this example, and how the pointer Λ is updated.

When reading the rules, keep in mind that the environment is defined as follows:

$$\text{Env}(N_\Psi, \tau_\Psi, \Psi, N_M, \tau_M, M, \mathbb{ID}_\Psi, \mathbb{ID}_M, \Lambda).$$

The first rule is:

```
rew <(Env(empty, empty, empty, empty, empty, empty, empty, empty, 0), N), (
    uint "companyNumber") = EXP(1) > .
```

As we can see, the resulting environment has the variable *companyNumber* stored in the address 0. Its type is *uint*, has value 1, and Λ has been increased to the next free address: 256.

Taking this environment as our input, we now declare the variable *bossIdentifier*.

```
rew <(Env("companyNumber" |-> 0, "companyNumber" |-> uint, 0 |-> CV(SV(1)),
     empty, empty, empty, "companyNumber", empty, 256), N), address "
    bossIdentifier" > .
```

The resulting environment has now stored both *companyNumber* and *bossIdentifier*, which has been initialised with the default value of the type *address*; 0. The value of Λ hass been updated to 416.

Now the mapping *workers* is declared.

```
rew <(Env(("bossIdentifier" |-> 256, "companyNumber" |-> 0), ("
    bossIdentifier" |-> address, "companyNumber" |-> uint), (0 |-> CV(SV(1)
    ), 256 |-> CV(SV(0))), empty, empty, empty, ("bossIdentifier", "
    companyNumber"), empty, 416), N), mapping(address => worker) "workers"
    > .
```

Note how despite the value of Λ was 416, the mapping has been stored in the address 512, and no value has been given to it. This is the expected behaviour, as we saw in the example from Chapter 6. The rest of the environment has been updated correctly.

The following instruction to be executed is the declaration of the last state variable; *shifts*, an empty dynamic array.

```
rew <(Env(("bossIdentifier" |-> 256, "companyNumber" |-> 0, "workers" |->
    512), ("bossIdentifier" |-> address, "companyNumber" |-> uint, "workers
    " |-> mapping(address => worker)), (0 |-> CV(SV(1)), 256 |-> CV(SV(0)))
    , empty, empty, empty, ("bossIdentifier", "companyNumber", "workers"),
    empty, 768), N), shift[] "shifts" > .
```

In this case, in the address given to the array, 768, the value 0 has been saved, which corresponds to the actual array size.

Now the first local variable is declared. We declare the variable *boss*, which is a *worker* struct. The variable is given a value; a list of four elements, one for each of its members.

```
rew <(Env(("bossIdentifier" |-> 256, "companyNumber" |-> 0, "shifts" |->
    768, "workers" |-> 512), ("bossIdentifier" |-> address, "companyNumber"
     |-> uint, "shifts" |-> shift[], "workers" |-> mapping(address =>
    worker)), (0 |-> CV(SV(1)), 256 |-> CV(SV(0)), 768 |-> CV(SV(0))),
    empty, empty, empty, ("bossIdentifier", "companyNumber", "shifts", "
    workers"), empty, 1024), N), (worker S "boss") = s(EXP(1) - EXP(false)
    - EXP(5) - "bossIdentifier" ) > .
```

Once the instruction has been processed, we can see how the environment has been updated accordingly, saving each of the values in its corresponding address: the value 1 has been saved in the address 1024, taking 256 bits, since it is an unsigned integer. In the address 1280 the bool *false* is stored, followed by the uint8 with value 5 in the address

1281. The last element is an address with value 0, since the variable *boss* had value 0, stored in the address 1289. Note how, despite the first empty slot in the storage is the address 1449, Λ has been updated to 1536, since structs are stored using full slots.

The last instruction corresponds to adding a key in the mapping.

```
rew <(Env(("boss" |-> 1024, "bossIdentifier" |-> 256, "companyNumber" |->
    0, "shifts" |-> 768, "workers" |-> 512), ("boss" |-> worker, "
    bossIdentifier" |-> address, "companyNumber" |-> uint, "shifts" |->
    shift[], "workers" |-> mapping(address => worker)), (0 |-> CV(SV(1)),
    256 |-> CV(SV(0)), 768 |-> CV(SV(0)), 1024 |-> CV(SV(1)), 1280 |-> CV(
    SV(false)), 1281 |-> CV(SV(5)), 1289 |-> CV(SV(0))), empty, empty,
    empty, ("boss", "bossIdentifier", "companyNumber", "shifts", "workers")
    , empty, 1536), N), "workers"["bossIdentifier"] = "boss" > .
```

To show the outcome of this rule, we have written the tuple returned by the execution step. We have divided the output in several lines for readability.

Note how the values have been saved in abstract addresses, starting from the address $\mathbf{HASH}(512 \cdot 0)$.

```
<(Env(
("boss" |-> 1024, "bossIdentifier" |-> 256, "companyNumber" |-> 0, "shifts"
     |-> 768, "workers" |-> 512),
("boss" |-> worker, "bossIdentifier" |-> address, "companyNumber" |-> uint,
     "shifts" |-> shift[], "workers" |-> mapping(address => worker)),
(0 |-> CV(SV(1)), 256 |-> CV(SV(0)), 768 |-> CV(SV(0)), 1024 |-> CV(SV(1)),
     1280 |-> CV(SV(false)), 1281 |-> CV(SV(5)), 1289 |-> CV(SV(0)), hash(
    concatena(512,CV(SV(0)))) |-> CV(SV(1)), 256 * (1 + floor((1 + 256 * (1
     + floor((256 + hash(concatena(512,CV(SV(0))))) / 256))) / 256)) |-> CV
    (SV(5)), 256 * (1 + floor((8 + 256 * (1 + floor((1 + 256 * (1 + floor
    ((256 + hash(concatena(512,CV(SV(0))))) / 256))) / 256))) / 256)) |->
    CV(SV(0)), 256 * (1 + floor((256 + hash(concatena(512,CV(SV(0))))) /
    256)) |-> CV(SV(false))),
empty, empty, empty,
("boss", "bossIdentifier", "companyNumber", "shifts", "workers"),
empty, 1536),
N),
(nil).List{CValor} >
```

8 Conclusion

Once we have finished showing the work we have done, we can see that on the one hand the semantics proposed here extend the big-step operational semantics consulted, trying to correct what we believed was incorrect or imprecise, and coming up with some new concepts, like the rigorous study of the storage. On the other hand, the implementation is currently incomplete, yet in our opinion, it could well lay the groundwork for the

implementation of the first functional Solidity semantics in Maude that we know of.

During the months I have spent developing this work, I have learnt the basics on blockchain technologies and Solidity and Maude languages; the first one with the aim of understanding it better in order to develop its formal semantics, and the latter in order to implement the semantics in code. The field in which I have learnt the most is semantics; previous to this work I only had some knowledge on semantics for simple languages like While, and having to provide rules for a much more complex language like Solidity, specially when addressing the memory system and the treatment of complex variables, like structs, arrays or mappings, has made me acquire a more complete understanding and familiarisation with operational semantics.

Another conclusion that can be reached after reading this work is how complex something like giving semantics for a language can get, despite restricting ourselves to a very narrow section of it, and how the amount of aspects to be covered increase significantly only from trying to broaden the scope slightly.

References

[1] Pedro Antonino and A. W. Roscoe. Formalising and verifying smart contracts with Solidifier: a bounded model checker for Solidity, 2020.

[2] Manuel Clavel, Francisco Durán, Steven Eker, Santiago Escobar, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, Rubén Rubio, and Carolyn Talcott. Maude Manual (version 3.0).

[3] Silvia Crafa, Matteo Di Pirro, and Elena Zucca. Is Solidity solid enough? In Andrea Bracciali, Jeremy Clark, Federico Pintore, Peter B. Rønne, and Massimiliano Sala, editors, *Financial Cryptography and Data Security*, pages 138–153, Cham, 2020. Springer International Publishing.

[4] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. A semantic framework for the security analysis of Ethereum smart contracts. In Lujo Bauer and Ralf Küsters, editors, *Principles of Security and Trust*, pages 243–269, Cham, 2018. Springer International Publishing.

[5] Ákos Hajdu and Dejan Jovanović. SMT-friendly formalization of the Solidity memory model. *Lecture Notes in Computer Science*, page 224–250, 2020.

[6] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu, and G. Rosu. Kevm: A complete formal semantics of the Ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 204–217, 2018.

[7] Jiao Jiao, Shuanglong Kan, Shang-Wei Lin, David Sanan, Yang Liu, and Jun Sun. Executable operational semantics of Solidity, 2018.

[8] Anastasia Mavridou, Aron Laszka, Emmanouela Stachtiari, and Abhishek Dubey. Verisolid: Correct-by-design smart contracts for Ethereum. In Ian Goldberg and Tyler Moore, editors, *Financial Cryptography and Data Security*, pages 446–465, Cham, 2019. Springer International Publishing.

[9] Ritesh Modi. *Solidity programming essentials: a beginners guide to build smart contracts for Ethereum and blockchain.* Packt Publishing, 2018.

[10] Koshik Raj. *Foundations of Blockchain: The pathway to crytocurrencies and decentralized blockchain applications.* Packt, 2019.

[11] Solidity documentation release 0.6.7.

[12] M. Szmigiera. Number of blockchain wallets 2019, Feb 2020.

[13] Alberto Verdejo and Narciso Martí-Oliet. Executable structural operational semantics in maude. *Journal of Logic and Algebraic Programming*, 67:226–293, 03 2004.

[14] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger byzantium version 7e819ec. *Ethereum project yellow paper*, 2019.

[15] Zheng Yang and Hang Lei. Lolisa: Formal syntax and semantics for a subset of the Solidity programming language. *ArXiv*, abs/1803.09885, 2018.

[16] Jakub Zakrzewski. Towards verification of Ethereum smart contracts: A formalization of core of Solidity. In Ruzica Piskac and Philipp Rümmer, editors, *VSTTE*, volume 11294 of *Lecture Notes in Computer Science*, pages 229–247. Springer, 2018.