# Towards an Operational Semantics for Solidity

Marco Crosara
Dept. of Computer Science
University of Verona, Italy
Email: marco.crosara@studenti.univr.it

Gabriele Centurino
Dept. of Computer Science
University of Verona, Italy
Email: gabriele.centurino@studenti.univr.it

Vincenzo Arceri
Dept. of Computer Science
University of Verona, Italy
Email: vincenzo.arceri@univr.it

***Abstract*—Solidity is a multi-paradigm programming language used for writing smart contracts on the Ethereum blockchain and offers a wide range of features, such as Ethereum transfers between contracts or wallets of normal users. Its specification is not formally defined, the behaviours of Solidity constructs are informally provided by its documentation, leading to misunderstandings and buggy code. Without a formal semantics, reasoning about programs becomes extremely hard, if not impossible. In this paper, we provide a first-step towards a formal operational semantics for Solidity, defining a memory model for the language, able to capture its main features.**

***Keywords–Programming Languages; Solidity; Semantics.***

## I. INTRODUCTION

We intend to define a complete semantics of a core of Solidity [1], but this is not a simple task. In order to reach this goal, we have to deal with an unusual actor: the blockchain [2]. Due to its presence, providing a formal semantics for Solidity [3] could result challenging for two reasons. The first one is relative to the frequently updating language, since Solidity continuously changes the constructs and mechanics of operations [4]. For this reason, in this work, we have chosen a specific version of the compiler: 0.5.10. The second one is that we have to deal with the Storage of the blockchain, that is separated from the memory of the Ethereum Virtual Machine (EVM). In this paper, we provide a first step toward a formal semantics of Solidity, modelling the memory and the interaction that happens between a smart contract and the blockchain. In the next section, we describe the Solidity language, the domains and the memory model used. In Section III we present its concrete semantics for some basic constructs and in Section IV we extend the semantics to contracts and functions. Finally, we provide some ideas for future related works.

## II. SOLIDITY

Solidity is the most popular language to write smart contracts on the Ethereum blockchain. Intuitively, a smart contract is a computer program designed to execute some actions when some condition is verified [2]. Solidity has been designed to offer a simple way to develop a smart contract and for this reason, it is strongly inspired by JavaScript. Unlike JavaScript, it is object-oriented and statically typed. When we deploy a contract on the blockchain, the Solidity code has to be executed by the EVM. Inside this environment, we have a set of instructions called opcodes that are encoded in byte code in order to have a more efficient store. Each opcode has a cost of execution, this is needed to prevent the execution of infinite loops or similar and to reward the miners who validate the transactions. This cost is expressed in unit of Gas and the price per unit is expressed in GWei, a fraction of an Ethereum token. 1 Ethereum (ETH) corresponds to $1 \times 10^{18}$ Wei, that are $1 \times 10^{9}$ GWei. For the sake of simplicity, we assume that any operation inside the blockchain has been equipped with enough gas to correctly end its execution. In our work, we chose not to handle transactions in memory, studying only the interaction that they have with the blockchain.

### A. Memory and Storage

Solidity provides three types of memory, namely Stack, used to hold local variables of primitive type (*uint256*, *bool*, etc.), Storage is a persistent memory and is a key-value store where keys and values are both 32 bytes, storing, for instance, state variables. Finally, Memory is a byte-array that contains data until the execution of the function, used to save, for example, function arguments. In this paper, we do not distinguish between Stack and Memory. According to the real model described, the evaluation of expressions and statements in our work is made considering the tuple $\sigma = \langle N_\rho, \rho, C, A \rangle$. We can split this tuple in two halves, namely Memory and Storage. The first one refers to the EVM Memory and the second one to the blockchain. $N_\rho$ e $\rho$ are respectively the Namespace and the link between address and values. Instead $A$ stands for Accounts and contains the balances of contract address and normal user address. $C$ stands for Contracts and contains, for any contract, the corresponding Storage and all the functions with the corresponding signature. Formally, we define the State $\sigma$, as follows.

– $N_\rho \in$ Memory is a function s.t. $N_\rho : \text{ID} \rightarrow \text{MLOC}$

```
contract Bank {
  uint money = 0;
  constructor () public payable {
    money = msg.value;
  }
  function sendEther () public payable {
    money += msg.value;
    if(money > 300000000000000000){//0.3 ETH
      msg.sender.transfer(money);
      money = 0;
    }
  }
  function () external payable {}
}
```

Figure 1. Example of a simple contract written in Solidity.

Copyright (c) IARIA, 2019. ISBN: 978-1-61208-755-9 

```
Solidity ::= (Contract)*
Contract ::= contract id { (St)* }
St     ::= Method | StateDef
StateDef ::= Type id ;
        | Type id = Exp ;
Type ::= uint | bool | address | address payable
Method ::= function id((Type id,)*) (Qualifier)*
        {(Stmt)*}
        | function()external payable { (Stmt)* }
        | constructor((Type id,)*) public |
          internal { (Stmt)* }
Qualifier ::= public | internal | external
        | private | returns( Type id )
BinOp ::= + | - | * | / | % | && | || | == | !=
        | > | < | >= | <=
UnOp ::= - | !

Stmt ::= ε
        | Type id (= Exp)? ;
        | if( Exp )Stmt (else Stmt)?
        | while( Exp ) Stmt
        | { (Stmt)* }
        | return (Exp)? ;
        | Exp ;
Exp  ::= Literal
        | id( (Exp ,)* )
        | id.transfer(Exp)
        | Exp BinOp Exp
        | UnOp Exp
        | id = Exp
        | id
Literal ::= n ∈ UINT | b ∈ BOOL | a ∈ ADDR | ã ∈ ADDR_P
```

Figure 2. Syntax of Solidity core.

- $\rho \in$ Memory is a function s.t. $\rho$ : MLOC $\rightarrow$ V with V = UINT $\cup$ BOOL $\cup$ ADDR $\cup$ $\text{ADDR}_\text{P}$
- $A \in$ Storage is a function s.t. $A$ : ADDR $\rightarrow$ UINT
- $C \in$ Storage is a function s.t. $C$ : ADDR $\rightarrow \langle \lambda, N_\mu, \mu \rangle$ Where $\lambda = \langle P, I, E, R \rangle$, $N_\mu$ : ID $\rightarrow$ SLOC and $\mu$ : SLOC $\rightarrow$ V

$\lambda$ contains contracts functions that are divided by access level: Public, Internal, External, PRivate $\langle P, I, E, R \rangle$, each element in $\lambda$ is also a function $\langle$ID, ForParams$\rangle \rightarrow$ BODY and ForParams is a list of $\langle$Type, ID$\rangle$ but for simplicity, sometimes we will refer to it with a string of the type (Type $id$,)*. BODY is a string with a sequence of statements: (Stmt)*. The qualifier of a function can be [4]:

- `public`: Public functions are part of the contract interface and can be either called internally or via messages.
- `internal`: Those functions and state variables can only be accessed internally (i.e., from within the current contract or contracts deriving from it), without using this.
- `external`: External functions are part of the contract interface, which means they can be called from other contracts and via transactions. An external function f cannot be called internally.
- `private`: Private functions and state variables are only visible for the contract they are defined in and not in derived contracts.

In this paper, we suppose that in each moment we have another namespace $N_\sigma$, which determines the last declaration of a variable, between Memory and Storage. Formally, it is always $N_\sigma = (\sigma.C(\dot{c}).N_\mu)[N_\rho]$ where $\dot{c}$ is the current contract address.

### B. Domains

The followings are the Solidity domains considered for this paper:

- $n \in$ UINT = { 0, 1, 2, ..., $2^{256} - 1$}: the domain of Unsigned Integers, corresponding to the *uint256* type in the Solidity language. We define two numbers, $\tilde{N} = 2^{256}$ and $\hat{N} = 2^{256} - 1$, where $\hat{N}$ is the max value that can be assigned.
- $a \in$ ADDR is the domain of Addresses. The addresses are used as unique identifier inside the blockchain: every contract, every user and every transaction has one. In Solidity the *address* type holds a 20 byte value (size of an Ethereum address), e.g., '0xbb9bc244d798123fde783fcc1c72d3bb8c189413'. The same address could be also declared as Address Payable, this is necessary to allow transfers of ETH on it, as we will explain later. The domain of payable address is $\text{ADDR}_\text{P}$ and with $\widetilde{a}$ we denote an element of it.
- $b \in$ BOOL = {`true`, `false`}: the domain of Booleans.
- $x \in$ ID is the domain of Identifiers. In Solidity, an identifier is a string with the pattern [a-zA-Z_$][a-zA-Z_$0-9]*. An ID element could be a variable name, a contract name or a function name.
- LOC: the domain of Locations. We can have two types of locations, namely Memory Locations (MLOC) and Storage Locations (SLOC). Hence we have LOC = MLOC $\cup$ SLOC s.t. MLOC $\cap$ SLOC = $\varnothing$.

In our work, we denote by type$(\sigma, x) \in$ {`uint`, `bool`, `address`, `address payable`} the type $x$ in $\sigma$, e.g., type$(\langle \{x \rightarrow l\}, \{l \rightarrow 5\}, C, A \rangle, x)$ = `uint`. We abuse notation denoting type$(\sigma, l)$, $l \in$ LOC, the type of a location.

*Address and Address Payable:* In our core, there are two ways to declare addresses and the difference is the keyword `payable`. A payable address can be the receiver of some ETH sent using a `transfer` or a `send` function in a smart contract. Trying transfer money to a non-payable address would result in a compiler error. Therefore, for example the `transfer` function could not be invoked on a non-payable address. In sight of this, we can state that the keyword payable is only used in order to force the developer to wisely choose which address should be able to receive ether or not. In our semantics, the meta-variables of address can be interchangeable with the one of address payable.

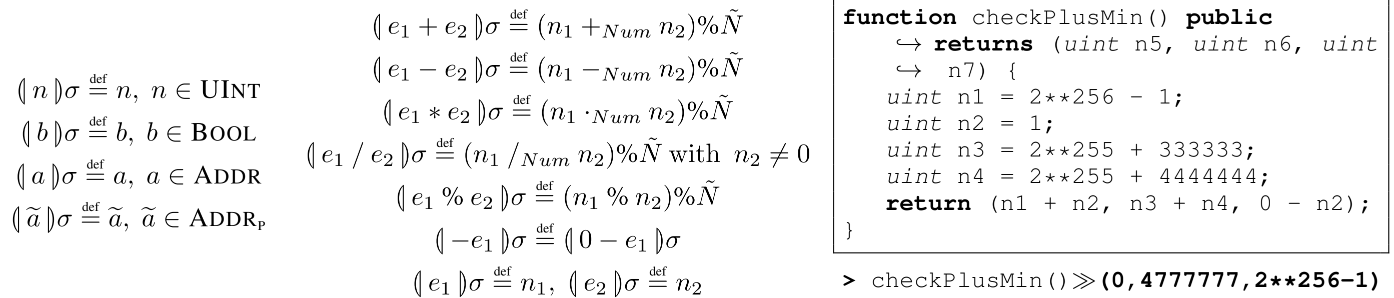


Figure 3. (a) Identity (b) Arithmetic expressions (c) Example of overflow.

### C. Environment access and Memory updating

In our semantics, when we need to access a tuple, for the sake of readability, we use the dots notation. For example, if $\sigma = \langle N_\rho, \rho, C, A\rangle$ we write $\sigma.C$ to access $C$ of $\sigma$. Note that, in some cases we will refer directly to $N_\rho, \rho, C, A, \lambda, N_\mu, \mu$ respectively instead of each ones with $\sigma$ prefixed: $\sigma.N_\rho, \sigma.\rho, \sigma.C$...

Updating of a single value in Memory:

$$\begin{gathered}\rho \in \text{Memory},\ l \in \text{MLoc},\\ k \in \text{V} = \text{UInt} \cup \text{Bool} \cup \text{Addr} \cup \text{Addr}_\text{P}\\ \rho[l \leftarrow k] = \rho' \in \text{Memory}\\ \Longleftrightarrow\\ \rho'(l) = k \ \text{ and } \ \forall l' \in \text{MLoc} \,.\, l' \neq l \,.\, \rho'(l') = \rho(l')\end{gathered}$$

Updating of a Memory with another Memory:

$$\begin{gathered}\rho, \rho' \in \text{Memory},\ \ l \in \text{MLoc},\ \ k \in \text{V}\\ Loc(\rho) = \{l \mid (l \mapsto k) \in \rho\}\\ \rho[\rho'] = \rho'' \in \text{Memory}\\ \Longleftrightarrow\\ \forall l \in Loc(\rho) \cup Loc(\rho') \,.\, \rho''(l) = \begin{cases}\rho'(l) & l \in Loc(\rho')\\ \rho(l) & \text{otherwise}\end{cases}\end{gathered}$$

Similarly, we can define Memory namespace update with single value ($N_\rho[x \leftarrow l]$) and update between Memory namespaces ($N_\rho[N'_\rho]$). The same is also true for $N_\mu, \mu, C, A$ update.

## III. Concrete semantics of Solidity

In this section, we define our core (Figure 2) and we provide a formal semantics for it [5]. We focus the attention on the standard constructs of programming languages, the more blockchain related constructs will be treated in Section IV. There will be also some examples that we will use to motivate the results. The semantics is captured by the function $(\!| \cdot |\!)$ that we will define in the next sections.

### A. Expressions Semantics

We denote the domain of expressions by $e \in$ Exp. In this section, we define the main semantics expressions of our Solidity core. The semantics of expressions is captured by the function $(\!| \cdot |\!)$ : Exp × State → V × State, that evaluates an expression in a State and returns a final value, with the previous State modified by the evaluation. For convenience when the State $\sigma$ does not change the evaluation, it returns only V. As we have already mentioned before, in our core we consider four primitive types: *bool*, *address* (payable and not), and *uint*. Each type has been defined as a domain of its values: UInt, Bool, Addr and Addr$_\text{P}$. According to this, we have four identity rules (Figure 3a). We also have other rules that refer to unary and binary operators. In Solidity, like in every other programming languages, we have many of them but we selected only the ones that have been defined in the syntax. Any numeric operator (Figure 3b) has a correspondent in the concrete. The rules consider the numeric overflow, indeed in UInt we can represent, like previously stated, the max value $\hat{N}$ and we will have an overflow using the next integer ($\tilde{N}$), this one is used by the rules through the modulo operator. A Solidity UInt overflow example is shown in Figure 3c.

The rules regarding *And*, *Or* and *Negation* for the binary operators (Figure 4a) are trivial. It is interesting to talk about *Equality* and *Inequality* operators among the different types. In these rules, we have the operator $\square \in \{==, !=\}$ and the operator $\boxdot$ that is the numeric counterpart of the first one. For example, if $==$ is the syntactic notation, $=_{Num}$ is the corresponding numeric operator. Each rule checks that the type of $e_1$ is the same of $e_2$ and returns a boolean value. An exception is the type Addr$_\text{P}$ that is equal to Addr for the reason previously specified. The next rule is similar to

$$(\!| e_1 \,\&\&\, e_2 |\!)\sigma \overset{\text{def}}{=} \begin{cases}\texttt{false} & (\!| e_1 |\!)\sigma \overset{\text{def}}{=} \texttt{false}\\ (\!| e_2 |\!)\sigma & (\!| e_1 |\!)\sigma \overset{\text{def}}{=} \texttt{true}\end{cases}$$

$$(\!| e_1 \,||\, e_2 |\!)\sigma \overset{\text{def}}{=} \begin{cases}\texttt{true} & (\!| e_1 |\!)\sigma \overset{\text{def}}{=} \texttt{true}\\ (\!| e_2 |\!)\sigma & (\!| e_1 |\!)\sigma \overset{\text{def}}{=} \texttt{false}\end{cases}$$

$$(\!| \,!e |\!)\sigma \overset{\text{def}}{=} \begin{cases}\texttt{true} & (\!| e |\!)\sigma \overset{\text{def}}{=} \texttt{false}\\ \texttt{false} & (\!| e |\!)\sigma \overset{\text{def}}{=} \texttt{true}\end{cases}$$

$$(\!| e_1 \,\square\, e_2 |\!)\sigma \overset{\text{def}}{=} \begin{cases} n_1 \boxdot_{Num} n_2 & (\!| e_1 |\!)\sigma \overset{\text{def}}{=} n_1 \wedge (\!| e_2 |\!)\sigma \overset{\text{def}}{=} n_2\\ b_1 \boxdot_{Bool} b_2 & (\!| e_1 |\!)\sigma \overset{\text{def}}{=} b_1 \wedge (\!| e_2 |\!)\sigma \overset{\text{def}}{=} b_2\\ a_1 \boxdot_{Adr} a_2 & (\!| e_1 |\!)\sigma \overset{\text{def}}{=} a_1 \wedge (\!| e_2 |\!)\sigma \overset{\text{def}}{=} a_2\\ \widetilde{a}_1 \boxdot_{Adr} \widetilde{a}_2 & (\!| e_1 |\!)\sigma \overset{\text{def}}{=} \widetilde{a}_1 \wedge (\!| e_2 |\!)\sigma \overset{\text{def}}{=} \widetilde{a}_2\\ a_1 \boxdot_{Adr} \widetilde{a}_2 & (\!| e_1 |\!)\sigma \overset{\text{def}}{=} a_1 \wedge (\!| e_2 |\!)\sigma \overset{\text{def}}{=} \widetilde{a}_2\end{cases}$$

$$(\!| e_1 \diamondsuit e_2 |\!)\sigma \overset{\text{def}}{=} (n_1 \ \diamondsuit_{Num} \ n_2) \in \{\texttt{true}, \texttt{false}\}$$

Figure 4. (a) Boolean expression semantics (b) Relational expression semantics.

the previous, but the operator $\Diamond \in \{>, <, >=, <=\}$ with the counterpart $\diamondsuit$ is only defined for numerical expressions. Afterwards, we will define other two rules regarding the semantics expressions.

*1) Assignment:* In this rule and the following ones, we suppose that $\dot{c}$ is the current contract address and that $\sigma = \langle N_\rho, \rho, C, A\rangle$.

$$(\!| x = e |\!)\sigma \stackrel{\text{def}}{=} \begin{cases} \langle g, \sigma'\rangle \text{ with } g \stackrel{\text{def}}{=} (\!| e |\!)\sigma & N_\sigma(x) \in \text{MLOC} \\ \langle g, \sigma''\rangle \text{ with } g \stackrel{\text{def}}{=} (\!| e |\!)\sigma & N_\sigma(x) \in \text{SLOC} \end{cases}$$
$$\sigma' = \langle N_\rho, \rho[N_\rho(x) \leftarrow g], C, A\rangle$$
$$\sigma'' = \langle N_\rho, \rho, C[\dot{c} \leftarrow \langle \lambda, N_\mu, \mu''\rangle], A\rangle$$
$$\text{with } \mu'' = \mu[N_\mu(x) \leftarrow g]$$
$$\text{if } N_\sigma(x) \neq \bot$$

The assignment in Solidity depends on the variable $x$ which we are referring to. If $N_\sigma(x) \in$ MLOC it means that the variable has been defined into the EVM Memory (potentially could exist an $x$ inside the Store). In this case, priority is given to the local variable and we only modified $\rho$ based on the address contained in $N_\rho$. Otherwise, if $N_\sigma(x) \in$ SLOC it means that it does not exist a local variable with that identifier. However, for the precondition rule $N_\sigma(x) \neq \bot$, there is always a global variable $x$, thus we modify $\mu$ associating the evaluation result of $e$ to the Storage address of $x$.

*2) Lookup:*

$$(\!| x |\!)\sigma \stackrel{\text{def}}{=} \begin{cases} \rho(N_\rho(x)) & N_\sigma(x) \in \text{MLOC} \\ C(\dot{c}).\mu(C(\dot{c}).N_\mu(x)) & N_\sigma(x) \in \text{SLOC} \end{cases}$$
$$\text{if } N_\sigma(x) \neq \bot$$

Like the previous rule, when in the code a variable $x$ is used, the returned value is determined with reference to the location where the last declaration happened. According to this, the value of $x$ in the Memory $\rho$ or in the Storage $\mu$ is returned.

### B. Statements Semantics

In this section, we define the formal semantics of Statements. Let denote by $s \in$ Stmt the sets of statements. With a slight abuse of notation, we denote the statement semantics evaluation with $(\!| \cdot |\!)$ : Stmt × State → State, that evaluates a statement in a State $\sigma$ and returns the State modified by the evaluation.

*1) Skip:*

$$(\!| \epsilon |\!)\sigma \stackrel{\text{def}}{=} \sigma \text{ where } \epsilon \text{ is the empty statement}$$

*2) Lacal Variable Declaration:*

$$(\!| \texttt{uint } x = e_1;\ |\!)\sigma \stackrel{\text{def}}{=} \langle N_\rho[x \leftarrow l], \rho[l \leftarrow (\!| e_1 |\!)\sigma], C, A\rangle$$
$$\text{with } (\!| e_1 |\!)\sigma \in \text{UINT}$$
$$l \in \text{MLOC fresh and } N_\rho(x) = \bot$$

The first semantics rule in this section is the empty statement, the following are regarding the variables declaration. The declaration of local and state variables is syntactically the same, so the correct rule is chosen accordingly to the position of statement. We distinguish if the declaration is inside the body of a function or directly inside the contract. The declaration of a local variable, differently from the only assignment, also modifies $N_\rho$. Therefore, a new location is added and the evaluated expression will be saved on it.

*3) State Variable Declaration:*

$$(\!| \texttt{uint } x = e_1;\ |\!)\sigma \stackrel{\text{def}}{=} \sigma' = \langle N_\rho, \rho, C', A\rangle$$
$$\text{with } C' = (C[\dot{c}.N_\mu(x) \leftarrow l])[\dot{c}.\mu(l) \leftarrow (\!| e_1 |\!)\sigma]$$
$$\text{with } (\!| e_1 |\!)\sigma \in \text{UINT},\ l \in \text{SLOC fresh}$$
$$\text{if } N_\mu(x) = \bot$$

The declaration of state variable is similar but in this case $N_\mu$ and $\mu$ are modified. In each case, there is a precondition: a variable with the same name must not be already declared. The rules for the other primitives types, which differ from *uint* are easily deducible for similarity.

*4) Declaration without initialisation:*

$$(\!| \texttt{uint } x;\ |\!)\sigma \stackrel{\text{def}}{=} (\!| \texttt{uint } x = 0;\ |\!)\sigma$$
$$(\!| \texttt{bool } y;\ |\!)\sigma \stackrel{\text{def}}{=} (\!| \texttt{bool } y = false;\ |\!)\sigma$$
$$(\!| \texttt{address } z;\ |\!)\sigma \stackrel{\text{def}}{=} (\!| \texttt{address } z = 0\text{x}0^{40};\ |\!)\sigma$$

Rules used for the declaration without initialisation can be defined as rewriting of the same rules with assignment. The value assigned to the variable is the default value of each primitive types. Other semantics rules related to constructs in our core in Figure 5a, are the one for `if` (rewrite of `if else`) and for `while`, where the single iteration is based on the rewrite of `if else`. Then, we have the semantics of block: after evaluating the statement inside the braces, the Memory of such evaluation is returned, preserving however the initial namespace $N_\rho$. This happens because the declaration made inside a block must not be considered as valid outside of it. Examples are presented in Figure 5b and Figure 5c. Finally, for the sequence of statements let's proceed evaluating the first statement. On the state returned we evaluate the next statement.

## IV. Concrete semantics of contracts

In this section, we provide the operational semantics for contracts and functions. A Solidity file has *sol* extension, it could contain some contracts, which are denoted by $c$. A file can be considered as a sequence of contracts $\texttt{C}$. We denote by $st$ a structure type and by $St$ a sequence of structure type. A $st$ could be a state variable or a function. In addition $\omega$ is used to denote the constructor of the contract.

*1) First:*

$$(\!| c_1\texttt{C} |\!)\sigma \stackrel{\text{def}}{=} (\!| \texttt{C} |\!)\sigma'' \text{ with } \sigma'' = \langle N_\rho, \rho, C', A'\rangle$$
$$\text{and } \sigma' = (\!| c_1 |\!)\sigma \stackrel{\text{def}}{=} \langle N'_\rho, \rho', C', A'\rangle$$
$$\text{with } N_\rho, \rho \text{ empty}$$

To evaluate a Solidity file we have to execute the sequence of contracts which it contains. We evaluate every contract on the state returned from the execution of the previous one, replacing however $N'_\rho$ and $\rho'$ with a new empty Memory $N_\rho, \rho$. Indeed the Memory of the EVM is not preserved from the execution of a contract to an another. Let's make a consideration now: $C(\dot{c}).\lambda.P$ and $C(\dot{c}).\lambda.E$ are visible to all other contracts, but to call a method of another contract it is necessary to create an instance of it, e.g., `MyContract mc = new MyContract();` and that does not

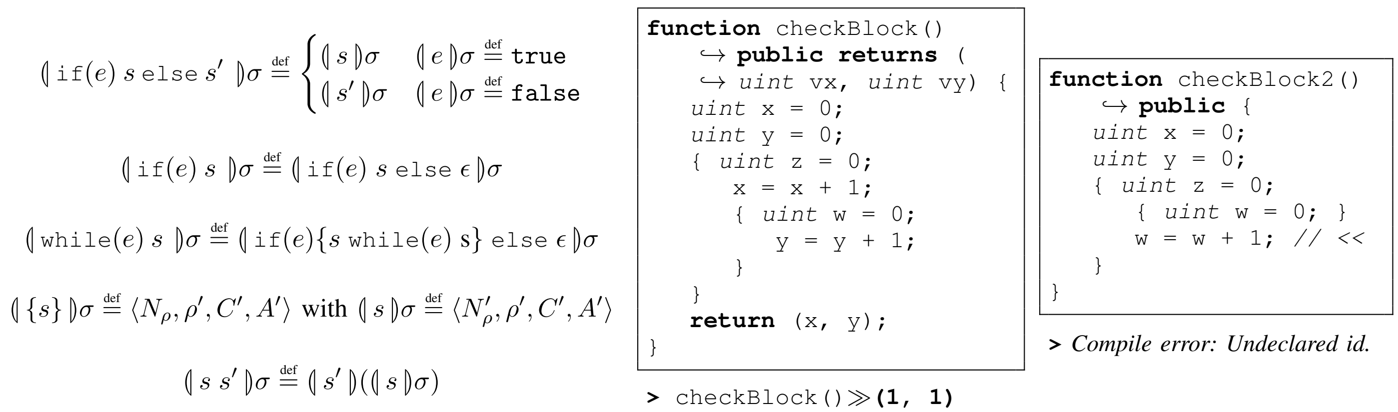


Figure 5. (a) IfElse, If, While, Block and Sequence of Stmt (b) Example of block: scoping of variables (c) Example of block: undeclared identifier.

exist in our core. Furthermore $C(\dot{c}).\lambda.I$ is directly visible to the contracts that derive from it, but to allow inheritance in Solidity we need the `is` construct that is not inside our core.

*2) Contract:*

$$(\!|\, \texttt{contract}\ \mathit{cname}\ \{St\}\, |\!)\sigma \stackrel{\text{def}}{=} \begin{cases} \sigma'' & \omega \in St \\ (\!|\, c'\, |\!)\sigma & \omega \notin St \end{cases}$$

$$c' = \texttt{contract}\ \mathit{cname}\ \{\texttt{constructor() public\{\}}\ St\}$$

$$\sigma'' \stackrel{\text{def}}{=} \begin{cases} (\!|\, C'.\lambda.P(\mathit{constructor}, *)\, |\!)\sigma' & C'.\lambda.P(\mathit{con...}, *) \neq \bot \\ (\!|\, C'.\lambda.I(\mathit{constructor}, *)\, |\!)\sigma' & C'.\lambda.I(\mathit{con...}, *) \neq \bot \end{cases}$$

$$\text{with } \sigma' = (\!|\, St\, |\!)\sigma \stackrel{\text{def}}{=} \langle N_\rho, \rho, C', A' \rangle$$

The evaluation of a contract if there is not any constructor, add, firstly, the default constructor. Afterwards, all the structure types contained are evaluated. After that, $C(\dot{c})$ is populated with functions and state variables of the contract. The final step is to execute the constructor code and return the evaluation of it.

*3) Function Declaration:* For simplicity, we suppose that functions with a return statement at the end of body, have the respective `returns`(Exp) qualifier in the function definition.

$$(\!|\, \texttt{function}\ \mathit{fname}\ (FP)\ \texttt{public}\ \{\text{BODY}\}\, |\!)\sigma \stackrel{\text{def}}{=} \sigma'$$

$$(\!|\, \texttt{function}\ \mathit{fname}\ (FP)\ \texttt{public returns}\ (rp)\ \{\text{BODY}\}\, |\!)\sigma \stackrel{\text{def}}{=} \sigma'$$

$$\sigma' = \langle N_\rho, \rho, C', A \rangle \text{ with } C' = C[\dot{c}.\lambda.P.\langle \mathit{fname}, FP \rangle \leftarrow \text{BODY}]$$

$$\text{if}$$

$$C(\dot{c}).\lambda.P(\langle \mathit{fname}, FP \rangle) = \bot, C(\dot{c}).\lambda.I(\langle \mathit{fname}, FP \rangle) = \bot,$$
$$C(\dot{c}).\lambda.E(\langle \mathit{fname}, FP \rangle) = \bot, C(\dot{c}).\lambda.R(\langle \mathit{fname}, FP \rangle) = \bot$$

The evaluation of a function declaration is the addition of the same to $C(\dot{c}).\lambda$. For internal and private qualifiers the rule is equivalent, with the modification of $I$, $E$ and $R$ respectively instead of $P$.

*4) Constructor Declaration:* A constructor is optional. Only one constructor for each contract can be defined, which means that overloading is not supported. In the code, no function with name '*constructor*' can be defined. Constructor functions can be either public or internal. If there is no constructor, the contract will assume the default empty constructor.

$$(\!|\, \texttt{constructor}\ (FP)\ \texttt{public}\ \{\text{BODY}\}\, |\!)\sigma \stackrel{\text{def}}{=}$$
$$(\!|\, \texttt{function}\ \mathit{constructor}\ (FP)\ \texttt{public}\ \{\text{BODY}\}\, |\!)\sigma$$
$$\text{with } \sigma.C(\dot{c}).\lambda.P(\mathit{constructor}, *) = \bot$$
$$\text{and } \sigma.C(\dot{c}).\lambda.I(\mathit{constructor}, *) = \bot$$

The constructor evaluation can be treated as a rewrite of a normal function declaration, with identifier the word '*constructor*'. We use this trick because the Solidity syntax does not allow naming a function '*constructor*'. The rule is the same for `internal` qualifier.

*5) Fallback Function Declaration:* A fallback function is a particular function that can be inside a contract. It has two mandatory characteristics: it has to be anonymous and does not have any arguments. It is executed whenever a function identifier does not match the available functions or if the contract receives plain Ether without any other data associated with the transaction. For this reason, it is good practice to make it payable, so that it can receive ETH sent erroneously. Consequently, in our core, we choose that the fallback function is always payable. The fallback function has only 2300 units of gas, leaving not much capacity to perform operations except basic logging.

$$(\!|\, \texttt{function}\ ()\ \texttt{external payable}\ \{\text{BODY}\}\, |\!)\sigma \stackrel{\text{def}}{=}$$
$$(\!|\, \texttt{function}\ \epsilon\ (\varnothing)\ \texttt{external}\ \{\text{BODY}\}\, |\!)\sigma$$
$$\text{with } \sigma.C(\dot{c}).\lambda.E(\epsilon, \varnothing) = \bot$$

The idea of the fallback function semantics is the same as the constructor one. The rule is given as rewrite of function declaration with $\epsilon$, namely the empty string, as name.

*6) Return:*

$$(\!|\, \texttt{return;}\, |\!)\sigma \stackrel{\text{def}}{=} \sigma' = \langle N_\rho[\mathit{return} \leftarrow l], \rho[l \leftarrow \epsilon], C, A \rangle$$
$$(\!|\, \texttt{return}\ e\texttt{;}\, |\!)\sigma \stackrel{\text{def}}{=} \sigma' = \langle N_\rho[\mathit{return} \leftarrow l], \rho[l \leftarrow (\!|\, e\, |\!)\sigma], C, A \rangle$$

The return statement is the last Stmt of function. It returns directly a value or an expression that must be evaluated. To transfer the return value to the caller, we save it in the Memory with the identifier '*return*'. The function call knows that, once the evaluation of the function is completed, the return value is stored in $N'_\rho(\rho'(\mathit{return}))$.

*7) Function Call:*

$$(\!|\, fname(e_1...e_n) \,|\!)\sigma \stackrel{\text{def}}{=} (\!|\, \text{BODY} \,|\!)\sigma' = \langle N'_\rho(\rho'(return)), \sigma'' \rangle$$
where $\sigma' = \langle N'_\rho, \rho', C, A \rangle$ s.t. $N'_\rho = \{(fp_i \leftarrow l_i) \mid \forall i \in [1, n]\}$,
$$\rho' = \{(l_i \leftarrow (\!|\, e_i \,|\!)\sigma_{i-1}) \mid \forall i \in [1, n], \sigma_0 = \sigma\}$$
$$l_i \in \text{MLOC fresh and } \sigma'' = \langle N_\rho, \rho, C', A' \rangle$$

$$\text{BODY} = \begin{cases} \texttt{Pbody} = C(\dot{c}).\lambda.P(fname, \forall i \mid t_i) & \texttt{Pbody} \neq \bot \\ \texttt{Ibody} = C(\dot{c}).\lambda.I(fname, \forall i \mid t_i) & \texttt{Ibody} \neq \bot \\ \texttt{Ebody} = C(\dot{c}).\lambda.E(fname, \forall i \mid t_i) & \texttt{Ebody} \neq \bot \\ \texttt{Rbody} = C(\dot{c}).\lambda.R(fname, \forall i \mid t_i) & \texttt{Rbody} \neq \bot \end{cases}$$
$$\text{where } t_i = \text{type}((\!|\, e_i \,|\!)\sigma_{i-1})$$

The semantics of a function call corresponds to the execution result of the function body. In particular, the function body must be executed in a state taking into account of the parameters passed to the function call, memory $\sigma'$. Then $C$, $A$ are unattached from $\sigma$ while $N'_\rho$ and $\rho'$ contains all the associations between actual and formal parameters. The return instruction saves, as previously said, the final value in $N'_\rho(\rho'(return))$. This one is returned to the caller with the Storage modified by the last evaluation and the Memory that the caller had before the call.

*8) Transfer:* We have chosen to implement the `transfer` function. This is not the only way that exists to transfer ETH between addresses, but is the most secure. In fact, there are also the `call` function, that is now deprecated, and the `send` function that can be still used but, contrary to the `transfer` function, when it fails, it simply returns false and does not propagate the exception. This behaviour can lead to unwanted errors and vulnerabilities [6]. Regarding the way we implemented the transfer function, we choose not to handle the exceptions. We studied three possible results of transfers:

– A transfer is done between two contracts with a correct amount of ETH and no fallback function is invoked.
– A transfer is done between two contracts with a correct amount of ETH and a fallback function is invoked.
– A transfer is done between two contracts with an incorrect amount of ETH and this lead to an error.

$$(\!|\, \widetilde{a}.\texttt{transfer}(\text{n}) \,|\!)\sigma \stackrel{\text{def}}{=} \sigma'$$
$$\sigma' = \begin{cases} \langle N_\rho, \rho, C, (A[\dot{c} \leftarrow A(\dot{c}) - n])[\widetilde{a} \leftarrow A(\widetilde{a}) + n] \rangle & \text{1st case} \\ \\ \langle N_\rho, \rho, C', A' \rangle \stackrel{\text{def}}{=} (\!|\, C(\widetilde{a}).\lambda.E(\epsilon, \varnothing) \,|\!)\sigma'' & \\ \mid \sigma'' = \langle N_\rho, \rho, C, A' \rangle & \text{2nd case} \\ \mid A' = (A[\dot{c} \leftarrow A(\dot{c}) - n])[\widetilde{a} \leftarrow A(\widetilde{a}) + n] & \\ \\ exception & \text{3rd case} \end{cases}$$

$$\begin{aligned} \text{1st case}: &\ A(\dot{c}) \geq n \wedge (C(\widetilde{a}) = \bot \vee \\ &\ (C(\widetilde{a}) \neq \bot \wedge C(\widetilde{a}).\lambda.E(\epsilon, \varnothing) = \bot)) \\ \text{2nd case}: &\ A(\dot{c}) \geq n \wedge C(\widetilde{a}) \neq \bot \wedge C(\widetilde{a}).\lambda.E(\epsilon, \varnothing) \neq \bot \\ \text{3rd case}: &\ A(\dot{c}) < n \end{aligned}$$

The three cases are described before. In the first and second case, we transfer the amount of ETH from $\dot{c}$ to $\widetilde{a}$, but in the second case the recipient is also a contract, therefore the returned Memory depends on the execution of $\widetilde{a}$ fallback function. In Figure 1 we propose a simple example of a contract that receives Ether and returns the full amount through the invocation of function `sendEther` when the contract balance is at least 0.3 ETH. The contract mentions the *msg* field that we have not covered in this core. It contains useful information of the transaction, e.g., the sender and, for ETH transfers, the amount sent.

## V. Conclusion And Future Works

In this paper, we have introduced a Solidity core and a formal semantics for it. This required us to introduce a first concept of an abstract memory model, that is able to run the code on the EVM. This model is also able to represent blockchain and its complex structure and behaviour. In order to extend our work, the next step is to create a more complete and meaningful core by adding the missing constructs. In this way, we will be able to provide a better representation of the contracts on the blockchain. At this stage, our core is enough to give a first idea and can provide the semantics of only basic contracts. As a future work, we plan to build a static analyzer, based on abstract interpretation [7], for the smart contracts written in Solidity.

## References

[1] K. Bhargavan et al., "Formal verification of smart contracts: Short paper," in Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2016, Vienna, Austria, October 24, 2016, T. C. Murray and D. Stefan, Eds. ACM, 2016, pp. 91–96, URL: https://doi.org/10.1145/2993600.2993611 [accessed: 2019-10-22].
[2] B. K. Mohanta, S. S. Panda, and D. Jena, "An overview of smart contract and use cases in blockchain technology," in 2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT), July 2018, pp. 1–4.
[3] J. Zakrzewski, "Towards verification of ethereum smart contracts: A formalization of core of solidity," in Verified Software. Theories, Tools, and Experiments - 10th International Conference, VSTTE 2018, Oxford, UK, July 18-19, 2018, Revised Selected Papers, ser. Lecture Notes in Computer Science, R. Piskac and P. Rümmer, Eds., vol. 11294. Springer, 2018, pp. 229–247.
[4] "Ethereum - Solidity documentation," 2019, URL: https://solidity.readthedocs.io/en/v0.5.10 [accessed: 2019-10-22].
[5] J. Jiao et al., "Executable operational semantics of solidity," CoRR, vol. abs/1804.01295, 2018, URL: http://arxiv.org/abs/1804.01295 [accessed: 2019-10-22].
[6] "King of the Ether Throne - Post-Mortem Investigation," 2016, URL: https://www.kingoftheether.com/postmortem.html [accessed: 2019-10-22].
[7] P. Cousot and R. Cousot, "Automatic synthesis of optimal invariant assertions: Mathematical foundations," SIGART Newsletter, vol. 64, 1977, pp. 1–12.
[8] S. Sahoo, A. M. Fajge, R. Halder, and A. Cortesi, "A hierarchical and abstraction-based blockchain model," Applied Sciences, vol. 9, no. 11, Jun. 2019, p. 2343, URL: http://dx.doi.org/10.3390/app9112343 [accessed: 2019-10-22].