Formal Methods in Security and Privacy (SS20) Prof. Matteo Maffei

Small-step semantics for *tinyEVM* bytecode

TU Wien

1 Overview

In the following, we present the simplified smart contract language *tinyEVM* bytecode and its formal semantics. This language is a simplification of the EVM bytecode language that is used for writing smart contracts for the cryptocurrency Ethereum. Even though *tinyEVM* bytecode only supports a limited set of bytecodes and shows a simplified semantics, it still serves for illustrating the main features of the EVM bytecode language.

2 Small-step Relation

In the following, we will formally describe the semantics of *tinyEVM* bytecode smart contract execution. As you learned in the lecture, initially the execution of a smart contract is initiated by an *external transaction*. This transaction is sent by an *external account* and targets a *contract account*. Upon this external transaction, the code of the contract account is executed (and possibly money is transferred). The semantics that we show here describes the execution of the code of such a contract account. As the code execution might again transfer money and call other contracts (we talk about *internal transactions* in this case), the state of an external transaction execution can be seen as a stack of execution states, where each execution state represents the state within the execution of one specific contract that was (potentially) paused as it called another contract. We refer to these stacks as *call stacks*.

Formally, we define a small step relation $\rightarrow$ that characterizes a single execution step of a transaction execution. We write $\Gamma \vDash S \rightarrow S'$ to denote that the call stack $S \in \mathbb{S}$ evolves under the transaction environment $\Gamma \in \mathcal{T}_{env}$ to the call stack $S' \in \mathbb{S}$. The transaction environment contains information about the timestamp of the block that the transaction is part of and does not change over code execution.

Language

tinyEVM bytecode is an assembler-like bytecode language that operates on a stack-based machine. Its instructions consist of the following (limited) set of opcodes (also referred to as *instructions* or *commands*):

$$\begin{aligned}\mathcal{O} = \{&\mathsf{ADD}, \mathsf{AND}, \mathsf{LE}, \mathsf{PUSH}x, \mathsf{POP}, \mathsf{MLOAD}, \mathsf{MSTORE}, \mathsf{SLOAD}, \mathsf{SSTORE}, \mathsf{TIMESTAMP}, \mathsf{BALANCE},\\ &\mathsf{INPUT}, \mathsf{ADDRESS}, \mathsf{GAS}, \mathsf{JUMP}pc, \mathsf{JUMPI}pc, \mathsf{RETURN}, \mathsf{STOP}, \mathsf{FAIL}, \mathsf{CALL} \mid pc \in \mathbb{N} \wedge x \in \mathbb{Z}\}\end{aligned}$$

The exact meaning of the different opcodes will become clear when reading the semantic rules (presented later) that characterize their effects.

Call stacks

In Figure 1 we give a full grammar for call stacks and transaction environments:

$$
\begin{array}{rlcll}
\text{Call stacks} & \mathbb{S} & \ni & S & := \mathit{EXC} :: S_{plain} \mid \mathit{HALT}(\sigma, d, g) :: S_{plain} \mid S_{plain} \\
\text{Plain call stacks} & \mathbb{S}_{plain} & \ni & S_{plain} & := (\mu, \iota, \sigma) :: S_{plain} \mid \epsilon \\
\text{Machine states} & M & \ni & \mu & := (\mathit{gas}, \mathit{pc}, m, s) \\
\text{Execution environments} & I & \ni & \iota & := (\mathit{actor}, \mathit{input}, \mathit{code}) \\
\text{Global states} & \Sigma & \ni & \sigma & \\
\text{Account states} & \mathbb{A} & \ni & \mathit{acc} & := (b, \mathit{code}, \mathit{stor}) \\
\text{Transaction environments} & \mathcal{T}_{env} & \ni & \Gamma & := \mathit{timestamp}
\end{array}
$$

$$
\text{Notations:} \qquad
\begin{array}{c}
g, \mathit{gas}, \mathit{pc}, b \in \mathbb{N} \\
d, \mathit{actor}, \mathit{input}, \mathit{timestamp} \in \mathbb{Z} \\
m \in \mathbb{Z} \to \mathbb{Z}, \quad s \in \mathcal{L}(\mathbb{Z}) \\
\mathit{code} \in \mathcal{L}(\mathcal{O}) \\
\Sigma = \mathbb{Z} \to \mathbb{A}
\end{array}
$$

Figure 1: Grammar for call stacks and transaction environments

Call stacks are formally characterized as lists of *regular execution states* (tuples of the form (μ, ι, σ)) that are optionally topped with a *terminal state* which can either be the exception state EXC or the halting state $\mathit{HALT}(\sigma, d, g)$. Halting states carry information about the execution state in which the system halted, more precisely the state of the system σ (also called *global state*) at this point, the return data $d \in \mathbb{Z}$ of the call and the remaining gas $g \in \mathbb{N}$.

In the following, we give a short description of the different components of the regular execution states, namely the machine state μ, the execution environment ι and the global state σ:

2.0.1 Machine state

The local machine state μ represents the state of the underlying stack machine used for execution. Formally it is represented by a tuple $(\mathit{gas}, \mathit{pc}, m, s)$ holding the amount of gas $\mathit{gas} \in \mathbb{N}$ available for execution, the program counter $\mathit{pc} \in \mathbb{N}$, the local memory $m : \mathbb{Z} \to \mathbb{Z}$ (a mapping from integer addresses to integer values), and the machine stack $s \in \mathcal{L}(\mathbb{Z})$ (a list of integer values).

2.0.2 Execution environment

The execution environment ι of an internal transaction is a tuple of static parameters $(\mathit{actor}, \mathit{input}, \mathit{code})$ to the transaction that, i.a., determine the code to be executed and the account in whose context the code will be executed. The execution environment incorporates the following components: the active account $\mathit{actor} \in \mathbb{Z}$ that is the account that is currently executing and whose account state will be affected when instructions for storage modification or money transfer are executed; the input data $\mathit{input} \in \mathbb{Z}$ given to the transaction; the code $\mathit{code} \in \mathcal{L}(\mathcal{O})$ (formally a list of opcodes) that is executed by the transaction.

2.0.3 Global state

We represent the global state of the Ethereum blockchain as a mapping from account addresses to accounts. For sake of simplicity, we assume that for all possible addresses, there exists an account Accounts are composed of a balance $b \in \mathbb{N}$, a persistent unbounded storage $\mathit{stor} \in \mathbb{Z} \to \mathbb{Z}$ (a mapping from integer addresses to integer values) and the account's code $\mathit{code} \in \mathcal{L}(\mathcal{O})$. External accounts carry an empty code which makes their storage inaccessible and hence irrelevant.

Comments

Note that all values that might be used for computation are assumed to range over integers $\mathbb{Z}$. There are only few components that we require to stay in the range of naturals ($\mathbb{N}$). These are gas values, balances and the program counter. The inference rules will ensure that these values stay in this range. On the first sight, it might seem slightly weird to also allow addresses and memory positions to range over $\mathbb{Z}$, but this treatment heavily simplifies the formulation of the inference rules. In the real EVM, values on the stack are simply 256-bit words that get interpreted in a signed or unsigned fashion depending on the way that are used. For the sake of presentation, we decided to work on mathematical objects here instead and to use a unique interpretation for all values.

Notations

In order to present the small-step rules in a concise fashion we introduce some notations for accessing and updating state. Most of the state components used in the formalization of the EVM execution configurations consist of tuples. For sake of better readability, instead of accessing tuple components using projection, we name the components according to the variable names and use a dot notation for accessing them. To differentiate component names from variable names, we typeset components in sans serifs font. For example, given $\mu \in M$, we write $\mu.\mathsf{gas}$ to access the first component of the tuple μ. Similarly, we use a simple update notation for components. E.g., instead of writing *let* $\mu = (gas, pc, m, s)$ *in* $(gas, pc + 1, m, s)$, we write $\mu[\mathsf{pc} \to \mu.\mathsf{pc} + 1]$. For the case of incrementing or decrementing numerical values we use the usual short cuts $+=$ and $-=$ and would for example write the example shown before as $\mu[\mathsf{pc} \mathrel{+}= 1]$.

For the global state we use a slightly different notation for accessing and updating. As the global state is a mapping from addresses to account, the account's state can be accessed by applying the address to the global state. For updating we introduce a simplifying notation:

$$\sigma\big\langle addr \to s\big\rangle := \lambda a.\, a = addr\,?\, s \;:\; \sigma(a)$$

Accessing bytecode For extracting the command that is currently executed, the instruction at position $\mu.\mathsf{pc}$ of the code code provided in the execution environment needs to be accessed. For sake of presentation, we define a function doing so:

Definition 1 (Currently executed command)**.** The currently executed command in the machine state μ and execution environment ι is denoted by $\omega_{\mu,\iota}$ and defined as follows:

$$\omega_{\mu,\iota} := \begin{cases} \iota.\mathsf{code}\,[\mu.\mathsf{pc}] & \mu.\mathsf{pc} < |\iota.\mathsf{code}| \\ \mathsf{FAIL} & \text{otherwise} \end{cases}$$

Note that this way of defining the currently executed commands makes sure that whenever a smart contract execution runs out of it's program counter range (as it e.g. does not explicitly ends with a STOP or RETURN), an exception is thrown.

Small-step rules

In the following, we will define the small-step relation by *inference rules*. Formally, the small-step relation is the smallest relation satisfying these rules. More practically this means that whenever a step $\Gamma \vDash S \to S'$ was performed then it must have been justified by one of the rules that we present.

Exception: low gas The first rule that we present is a universal one that holds independently of the opcode that is executed: we always throw an exception if the remaining gas (as specified in the machine state) in not sufficient for performing an operation. As we assume constant gas costs of 1 uni for all commands, it is enough to specify this behaviour by the following rule:

$$\frac{\mu.\mathsf{gas} < 1}{\Gamma \vDash (\mu, \iota, \sigma) :: S \rightarrow \mathit{EXC} :: S}$$

Binary stack operations We start by giving the rules for binary stack operations. In this category we consider: ADD, AND and LE. All of these instructions alter only the local stack and gas and their only difference consists of the operations performed. Still, we spell out the details for the different instructions here.

$$\frac{\omega_{\mu,\iota} = \mathsf{ADD} \qquad \mu.\mathsf{gas} \geq 1 \qquad \mu.\mathsf{s} = a :: b :: s \qquad \mu' = \mu[\mathsf{s} \rightarrow (a + b) :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 1]}{\Gamma \vDash (\mu, \iota, \sigma) :: S \rightarrow (\mu', \iota, \sigma) :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{AND} \qquad \mu.\mathsf{gas} \geq 1 \\ \mu.\mathsf{s} = a :: b :: s \qquad c = (a > 0 \wedge b > 0)\,?\,1 \,:\, 0 \qquad \mu' = \mu[\mathsf{s} \rightarrow c :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 1]\end{array}}{\Gamma \vDash (\mu, \iota, \sigma) :: S \rightarrow (\mu', \iota, \sigma) :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{LE} \\ \mu.\mathsf{gas} \geq 1 \qquad \mu.\mathsf{s} = a :: b :: s \qquad c = (a \leq b)\,?\,1 \,:\, 0 \qquad \mu' = \mu[\mathsf{s} \rightarrow c :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 1]\end{array}}{\Gamma \vDash (\mu, \iota, \sigma) :: S \rightarrow (\mu', \iota, \sigma) :: S}$$

We summarize the exception cases of these tree rules in one using the slightly sloppy $\vee$ notation. Technically, a separate inference rule for each instruction would be needed.

$$\frac{(\omega_{\mu,\iota} = \mathsf{ADD} \;\vee\; \omega_{\mu,\iota} = \mathsf{AND} \;\vee\; \omega_{\mu,\iota} = \mathsf{LE}) \qquad |\mu.\mathsf{s}| < 2}{\Gamma \vDash (\mu, \iota, \sigma) :: S \rightarrow \mathit{EXC} :: S}$$

For increasing the expressiveness of our language, these instructions can easily extended by more operations (including also unary ones). However, as they do not provide any further interesting insights about the semantics, we omit them here.

Accessing the execution environment There are some simple instructions for accessing parts of the execution environment such as the addresses of the executing account and the data given as input to the call.

$$\frac{\omega_{\mu,\iota} = \mathsf{ADDRESS} \qquad \mu.\mathsf{gas} \geq 1 \qquad \mu' = \mu[\mathsf{s} \rightarrow \iota.\mathsf{actor} :: \mu.\mathsf{s}][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 1]}{\Gamma \vDash (\mu, \iota, \sigma) :: S \rightarrow (\mu', \iota, \sigma) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{INPUT} \qquad \mu.\mathsf{gas} \geq 1 \qquad \mu' = \mu[\mathsf{s} \rightarrow \iota.\mathsf{input} :: \mu.\mathsf{s}][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 1]}{\Gamma \vDash (\mu, \iota, \sigma) :: S \rightarrow (\mu', \iota, \sigma) :: S}$$

Accessing the transaction environment

$$\frac{\omega_{\mu,\iota} = \mathsf{TIMESTAMP} \qquad \mu.\mathsf{gas} \geq 1 \qquad \mu' = \mu[\mathsf{s} \rightarrow \Gamma.\mathsf{timestamp} :: \mu.\mathsf{s}][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 1]}{\Gamma \vDash (\mu, \iota, \sigma) :: S \rightarrow (\mu', \iota, \sigma) :: S}$$

Accessing the global state The following operation pushes the balance of the specified contract to the stack.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{BALANCE} \qquad \mu.\mathsf{s} = a :: s \\ \mu.\mathsf{gas} \geq 1 \qquad \sigma(a) = (\mathit{balance}, \mathit{stor}, \mathit{code}) \qquad \mu' = \mu[\mathsf{s} \rightarrow \mathit{balance} :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 1]\end{array}}{\Gamma \vDash (\mu, \iota, \sigma) :: S \rightarrow (\mu', \iota, \sigma) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{BALANCE} \qquad |\mu.\mathsf{s}| < 1}{\Gamma \vDash (\mu, \iota, \sigma) :: S \rightarrow \mathit{EXC} :: S}$$

Stack operations There are simple instructions for popping values from the stack and pushing values to the stack. In the case of the PUSH instruction, the value to be pushed is provided as an argument to the opcode.

$$\frac{\omega_{\mu,\iota} = \mathsf{POP} \qquad \mu.\mathsf{gas} \geq 1 \qquad \mu.\mathsf{s} = a :: s \qquad \mu' = \mu[\mathsf{s} \rightarrow s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 1]}{\Gamma \vDash (\mu, \iota, \sigma) :: S \rightarrow (\mu', \iota, \sigma) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{POP} \qquad |\mu.\mathsf{s}| < 1}{\Gamma \vDash (\mu, \iota, \sigma) :: S \rightarrow \mathit{EXC} :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{PUSH}x \qquad \mu.\mathsf{gas} \geq 1 \qquad \mu' = \mu[\mathsf{s} \rightarrow x :: \mu.\mathsf{s}][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 1]}{\Gamma \vDash (\mu, \iota, \sigma) :: S \rightarrow (\mu', \iota, \sigma) :: S}$$

Jumps The JUMPi command updates the program counter to i (specified in the bytecode). A well-formed smart contract is expected only to contain jump destinations within the range of the contract instructions (this is a property that can easily be checked statically).

$$\frac{\omega_{\mu,\iota} = \mathsf{JUMP}i \qquad \mu.\mathsf{gas} \geq 1 \qquad \mu' = \mu[\mathsf{s} \rightarrow s][\mathsf{pc} \rightarrow i][\mathsf{gas} \mathrel{-}= 1]}{\Gamma \vDash (\mu, \iota, \sigma) :: S \rightarrow (\mu', \iota, \sigma) :: S}$$

The conditional jump command JUMPIi conditionally jumps to position i depending on b.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{JUMPIi} \\ \mu.\mathsf{gas} \geq 1 \qquad \mu.\mathsf{s} = b :: s \qquad j = (b = 0)\,?\,\mu.\mathsf{pc} + 1 \,:\, i \qquad \mu' = \mu[\mathsf{s} \rightarrow s][\mathsf{pc} \rightarrow j][\mathsf{gas} \mathrel{-}= 1]\end{array}}{\Gamma \vDash (\mu, \iota, \sigma) :: S \rightarrow (\mu', \iota, \sigma) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{JUMPIi} \qquad |\mu.\mathsf{s}| < 1}{\Gamma \vDash (\mu, \iota, \sigma) :: S \rightarrow \mathit{EXC} :: S}$$

Local memory operations The MLOAD command reads a value from the local memory at memory address a and pushes it to the stack.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{MLOAD} \\ \mu.\mathsf{gas} \geq 1 \qquad \mu.\mathsf{s} = a :: s \qquad v = \mu.\mathsf{m}(a) \qquad \mu' = \mu[\mathsf{s} \to v :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 1]\end{array}}{\Gamma \vDash (\mu, \iota, \sigma) :: S \to (\mu', \iota, \sigma) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{MLOAD} \qquad |\mu.\mathsf{s}| < 1}{\Gamma \vDash (\mu, \iota, \sigma) :: S \to \mathit{EXC} :: S}$$

The MSTORE command writes a value b given at the stack to address a in the local memory.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{MSTORE} \\ \mu.\mathsf{s} = a :: b :: s \qquad \mu.\mathsf{gas} \geq 1 \qquad \mu' = \mu[\mathsf{m} \to \mu.\mathsf{m}[a \to b]][\mathsf{s} \to s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 1]\end{array}}{\Gamma \vDash (\mu, \iota, \sigma) :: S \to (\mu', \iota, \sigma) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{MSTORE} \qquad |\mu.\mathsf{s}| < 2}{\Gamma \vDash (\mu, \iota, \sigma) :: S \to \mathit{EXC} :: S}$$

Persistent storage operations The SLOAD command reads the executing account's persistent storage at position a.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{SLOAD} \\ \mu.\mathsf{gas} \geq 1 \qquad \mu.\mathsf{s} = a :: s \qquad \mu' = \mu[\mathsf{s} \to (\sigma(\iota.\mathit{addr}).\mathsf{stor})(a) :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 1]\end{array}}{\Gamma \vDash (\mu, \iota, \sigma) :: S \to (\mu', \iota, \sigma) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{SLOAD} \qquad |\mu.\mathsf{s}| < 1}{\Gamma \vDash (\mu, \iota, \sigma) :: S \to \mathit{EXC} :: S}$$

The SSTORE command stores the value b in the executing account's persistent storage at position a.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{SSTORE} \qquad \mu.\mathsf{gas} \geq 1 \qquad \mu.\mathsf{s} = a :: b :: s \\ \mu' = \mu[\mathsf{s} \to s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 1] \qquad \sigma' = \sigma\big\langle \iota.\mathit{addr} \to \iota.\mathit{addr}[\mathsf{stor} \to \sigma(\iota.\mathit{addr}).\mathsf{stor}[a \to b]]\big\rangle\end{array}}{\Gamma \vDash (\mu, \iota, \sigma) :: S \to (\mu', \iota, \sigma') :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{SSTORE} \qquad |\mu.\mathsf{s}| < 2}{\Gamma \vDash (\mu, \iota, \sigma) :: S \to \mathit{EXC} :: S}$$

Accessing the machine state

$$\frac{\omega_{\mu,\iota} = \mathsf{GAS} \qquad \mu.\mathsf{gas} \geq 1 \qquad \mu' = \mu[\mathsf{s} \to \mu.\mathsf{gas} :: \mu.\mathsf{s}][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 1]}{\Gamma \vDash (\mu, \iota, \sigma) :: S \to (\mu', \iota, \sigma) :: S}$$

Halting instructions The execution of a RETURN interprets the top element of the stack as return value and records it in the halting state in order to potentially propagate it to the caller.

$$\frac{\omega_{\mu,\iota} = \mathsf{RETURN} \qquad \mu.\mathsf{s} = d :: s \qquad \mu.\mathsf{gas} \geq 1 \qquad g = \mu.\mathsf{gas} - 1}{\Gamma \vDash (\mu, \iota, \sigma) :: S \rightarrow \mathit{HALT}(\sigma, g, d) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{RETURN} \qquad |\mu.\mathsf{s}| < 1}{\Gamma \vDash (\mu, \iota, \sigma) :: S \rightarrow \mathit{EXC} :: S}$$

The execution of a STOP command halts execution propagating zero as return value to the caller.

$$\frac{\omega_{\mu,\iota} = \mathsf{STOP} \qquad g = \mu.\mathsf{gas} - 1}{\Gamma \vDash (\mu, \iota, \sigma) :: S \rightarrow \mathit{HALT}(\sigma, g, 0) :: S}$$

The FAIL instruction causes a direct entering of the exception state.

$$\frac{\omega_{\mu,\iota} = \mathsf{FAIL}}{\Gamma \vDash (\mu, \iota, \sigma) :: S \rightarrow \mathit{EXC} :: S}$$

Calling The CALL command initiates a the execution of a (potentially different) account's code. To this end, it gets as parameters the address *to* of the destination account and the value *va* to be transferred to the destination account. Additionally the input data for the called code is specified (by *id*) and the address where the return value of the call shall to be written to in local memory (specified by *oa*) . If the balance of the calling account $\iota.\mathsf{actor}$ is sufficient to transfer *va* the recipient *to* gets the value *va* transferred from the calling account $\iota.\mathsf{actor}$. The input data input to the call is read from the stack and written to the execution environment. Additionally the execution environment is updated with the code to be executed (that is the code of the called account). The execution of the called code then starts in the updated execution environment and with an empty machine state.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{CALL} \qquad \mu.\mathsf{s} = \mathit{to} :: \mathit{va} :: \mathit{id} :: \mathit{oa} :: s \qquad \mu.\mathsf{gas} \geq 1 \\ \sigma(\iota.\mathsf{actor}).\mathsf{b} \geq \mathit{va} \geq 0 \qquad \sigma' = \sigma\langle \mathit{to} \rightarrow \sigma(\mathit{to})[\mathsf{b} \mathrel{+}= \mathit{va}]\rangle\langle \iota.\mathsf{actor} \rightarrow \sigma(\iota.\mathsf{actor})[\mathsf{b} \mathrel{-}= \mathit{va}]\rangle \\ \mu' = (\mu.\mathsf{gas} - 1, 0, \lambda x.\, 0, \epsilon) \qquad \iota' = (\mathit{to}, \mathit{id}, \sigma(\mathit{to}).\mathsf{code})\end{array}}{\Gamma \vDash (\mu, \iota, \sigma) :: S \rightarrow (\mu', \iota', \sigma') :: (\mu, \iota, \sigma) :: S}$$

Note that here, ϵ denotes the empty stack and $\lambda x.0$ denotes the function mapping every value to 0.

If the executing account $\iota.\mathsf{actor}$ does not hold the amount of wei specified to be transferred by the CALL instruction (*va*), the call does not get executed. In the small step semantics this is modelled by throwing an exception.

$$\frac{\omega_{\mu,\iota} = \mathsf{CALL} \qquad \mu.\mathsf{s} = \mathit{to} :: \mathit{va} :: \mathit{id} :: \mathit{oa} :: s \qquad \mu.\mathsf{gas} \geq 1 \qquad \mathit{va} > \sigma(\iota.\mathsf{actor}).\mathsf{balance}}{\Gamma \vDash (\mu, \iota, \sigma) :: S \rightarrow \mathit{EXC} :: S}$$

If the amount of wei specified to be transferred by the CALL instruction (*va*) is negative, an exception is thrown:

$$\frac{\omega_{\mu,\iota} = \mathsf{CALL} \qquad \mu.\mathsf{s} = \mathit{to} :: \mathit{va} :: \mathit{id} :: \mathit{oa} :: s \qquad \mu.\mathsf{gas} \geq 1 \qquad \mathit{va} < 0}{\Gamma \vDash (\mu, \iota, \sigma) :: S \rightarrow \mathit{EXC} :: S}$$

Also the exception is thrown if the stack size is less than 4:

$$\frac{\omega_{\mu,\iota} = \mathsf{CALL} \qquad \mu.\mathsf{s} = \mathit{to} :: \mathit{va} :: \mathit{id} :: \mathit{oa} :: s \qquad |\mu.\mathsf{s}| < 4}{\Gamma \vDash (\mu, \iota, \sigma) :: S \ \rightarrow \mathit{EXC} :: S}$$

For returning from a call, there are several options:

1. The execution of the called code ends with RETURN. In this case the call was successful. The stack of the caller specifies the address in the local memory that contains the return value. The return value is copied to the caller's local memory as specified on the caller's stack and the execution proceeds in the global state left by the callee. To indicate success 1 is written to the caller's stack.
2. The execution of the called code ends with STOP. This case is similar to RETURN only the return value is constantly 0.
3. The execution of the called code ends with an exception. In this case the remaining arguments are removed from the caller's stack and instead 0 is written to the caller's stack.

As the first two cases can be treated analogously, we just need two rules for returning from a call.

$$\frac{\mu.\mathsf{s} = \mathit{to} :: \mathit{va} :: \mathit{id} :: \mathit{oa} :: s \qquad \mu' = \mu[\mathsf{s} \rightarrow 1 :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \rightarrow \mathit{gas}][\mathsf{m} \rightarrow \mu.\mathsf{m}[\mathit{oa} \rightarrow d]]}{\Gamma \vDash \mathit{HALT}(\sigma', \mathit{gas}, d) :: (\mu, \iota, \sigma) :: S \ \rightarrow (\mu', \iota, \sigma') :: S}$$

$$\frac{\mu.\mathsf{s} = \mathit{to} :: \mathit{va} :: \mathit{id} :: \mathit{oa} :: s \qquad \mu' = \mu[\mathsf{s} \rightarrow 0 :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \rightarrow 0]}{\Gamma \vDash \mathit{EXC} :: (\mu, \iota, \sigma) :: S \ \rightarrow (\mu', \iota, \sigma) :: S}$$