A EVM-light model of the Mangrove Protocol.

A. Husson, J. Krivine

Abstract

This paper aims at modelling some core design principles of Mangrove, an order-book based decentralised protocol running on the Ethereum Virtual Machine (EVM) [3]. It assumes some familiarity with the Ethereum ecosystem. After a brief presentation of the Mangrove's features we are interested in modelling, we introduce an EVM-light that serves as operational semantics able to execute simple models of smart contracts. The final part of the paper is dedicated to the formal modelling of Mangrove in EVM-light and to proving that the model is able to cope with important attack vectors.

1 Mangrove

1.1 General principle

Mangrove [1] is a two sided protocol whose users are called *Takers* on the one hand and *Makers* on the other. Makers are typically arbitrary smart contracts that can promise a certain quantity N_O of *outbound* assets on Mangrove in exchange of N_I *inbound* assets. Assets are assumed to be managed by smart contracts that implement ERC20-like interfaces [2]. These promises are called (maker) *offers* and the *owner* of an offer is the address of the Maker that posted the promise. The ratio N_I/N_O is the (Maker) price that is used to sort offers: on a given outbound/inbound offer list, the first and best offer is the one with the lowest price. The present paper does not address the issue of price representation in the EVM. We assume here that arbitrary precision can be used.

Mangrove presents two main public functions: `newOffer` and `marketOrder`. The first one is used by Maker contracts to publish a new offer on Mangrove, while the second is called by Takers who wish to swap a certain amount of inbound tokens against as many offers as possible, as long as the price of the offers consumed stays below a *limit* specified by the taker. Recall that offers are consumed in the order induced by the offer list, from best to worst.

The innovation of Mangrove relies on its ability to perform calls to arbitrary smart contracts during the execution of the (Taker initiated) market order transaction: each time an offer is matched against a Taker market order, the Maker contract that posted the offer is called with an amount of gas that was requested when the offer was made. Hence, the `newOffer` function of Mangrove can be modelled as:

$$\texttt{newOffer}(\mathit{outbound}, \mathit{inbound}, \mathit{price}, \mathit{gasreq})$$

where *outbound* and *inbound* are the addresses of the ERC20 that manage respectively the inbound and outbound assets of the offer, *price* is the Maker price at which it is willing to exchange tokens, and *gasreq* is the amount of gas units that the Maker requires to execute its offer when called by Mangrove. If two offers have the same price, offers that were inserted first take precedence. Importantly, a Maker that wishes to call `newOffer` needs to have deposited, prior to calling the function, a certain amount of ether on Mangrove, which can be retrieved by the Maker when the offer is withdrawn. This deposit is called a *provision* and is used by Mangrove to penalise the Maker if its offer fails to deliver when called (see Section 1.3). At worst the gas spent by the offer is *gasreq*, so the provision required by Mangrove is $(\mathit{gasreq} + \mathit{gasbase}) * \mathit{gasprice}$, where *gasbase* is a protocol constant and *gasprice* is a protocol variable that covers a reasonable[1] gas price on the blockchain that hosts the Mangrove smart contract. Making sure that the amount of Takers' compensations is adequate is not studied in the present work, however we will prove that a compensation is indeed sent to the Taker if and only if the Maker's offer is to blame.

On the Taker's side, a call to Mangrove's `marketOrder` can be modelled as:

$$\texttt{marketOrder}(\mathit{outbound}, \mathit{inbound}, \mathit{outbound\_volume}, \mathit{limit\_price}, \mathit{limit\_gasreq})$$

[1]Typically an upper bound of the average gas price observed over a certain number of blocks.

where *outbound* and *inbound* identifies the traded assets, and *outbound_volume* is the number of inbound assets that the Taker expects for a *complete fill*. A Taker *partial fill* occurs when the Taker receives a volume v of outbound tokens with $0 < v <$ *outbound_volume*. The final arguments *limit_price* and *limit_gasreq* are respectively a price limit for the market order (see Section 1.2) and a bound on the amount of gas that can be spent on failing offers (see Section 1.3).

1.2 Offer execution

Suppose a Taker triggers a market order with the call:

$$\texttt{mgv.marketOrder}(o, i, V, l_p, l_g)$$

where `mgv` is the address of a deployed Mangrove contract. The protocol will select the current best promise of the (o, i) offer list. If the price of this offer is less or equal to l_p, Mangrove attempts to call the offer owner on a reserved `makerExecute` function with as many gas unit as the offer requires when posted. After a successful termination of `makerExecute` (i.e with no revert), Mangrove will attempt to transfer the promised outbound tokens from the Maker contract to itself, using the ERC20 public function `transferFrom`. If the Taker has enough gas and the amount of received outbound tokens is less than V (which includes the case where no outbound tokens were received), Mangrove will iterate the process on the new best offer of the list if its price is lower than l_p. If no such offer exists, Mangrove returns the accumulated outbound tokens to the Taker.

Notice that the Maker contract is given an amount of gas to perform *arbitrary* calls, which may be used to fetch the promised outbound tokens anywhere on chain, decide to renege on the trade (see Section 1.3), or actually decide to try to perform a reentrant attack on the protocol (see Section 1.4). The Figure 1 shows an example of a market order transaction.

1.3 Offers that fail to deliver

Because Maker contracts are not whitelisted, the Mangrove protocol needs to take care of *fail to deliver* scenarios. An offer owned by a Maker contract `mkr` fails to deliver if any on the following occurs:

1. Mangrove fails to transfer the Taker's inbound tokens to `mkr`.
2. The call to `mkr.makerExecute` reverts.
3. The call to `mkr.makerExecute` returns but Mangrove fails to transfer the promised assets from `mkr` to itself.

Case 1 occurs typically when the address `mkr` is blacklisted by the ERC20 in charge of maintaining the inbound token balance. Case 2 occurs for many reasons, including reverting on purpose in order to renege on trade. Another typical case is `mkr` reverting with an *out of gas* exception, which occurs when the *gasreq* argument of `newOffer` was underestimated by the Maker. Case 3 occurs when `mkr` has not approved Mangrove to call `transferFrom` on the ERC20 contract managing the outbound token. It may also occur if the `mkr` contract does not have the promised funds.

Importantly, when an offer fails to deliver, Mangrove is able to penalise its owner using the *provision* that was deposited by the Maker before calling the `newOffer` function.

1.4 Attack vectors

We are interested in showing that Mangrove's protocol effectively shields users from several important attack vectors:

Non compliance with Maker offers. Maker contracts that are called by Mangrove might not have full knowledge about the price and volume that they promised when posting their offers. Indeed recording such values would incur important gas cost, and Maker contracts should be able to trust Mangrove's data when called on `makerExecute`.

Similarly, Maker contracts that fail to deliver will compensate the Taker for the loss of gas. Since running out of gas during `makerExecute` will make offer fail to deliver, on must make sure that `makerExecute` is never called with less gas than required.

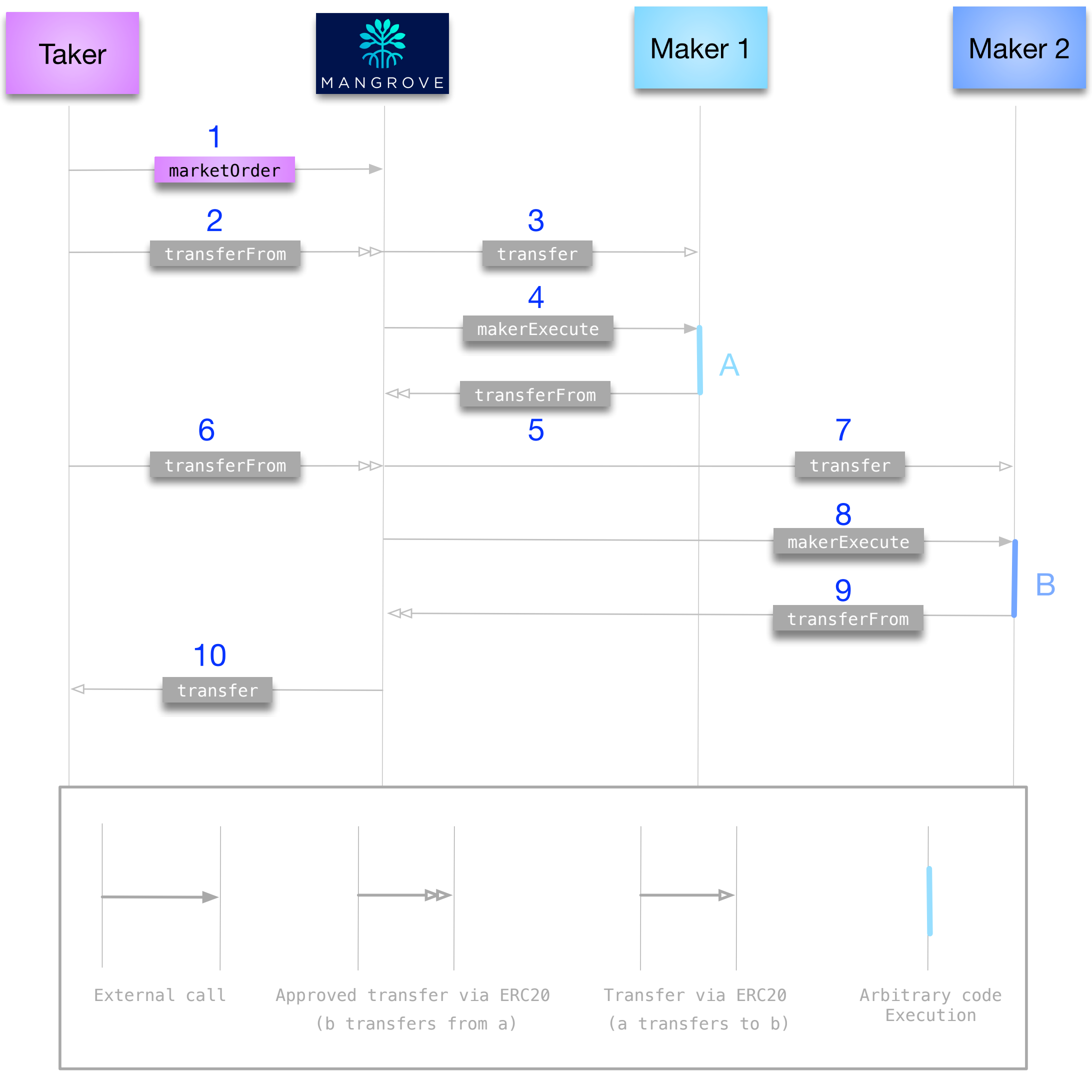

Figure 1: (1) A market order on Mangrove initiated by Taker. (2) Mangrove transfers inbound asset from Taker to itself via a call to the `transferFrom` function of the ERC20 in charge of the inbound asset. (3) The asset is then transferred to Maker 1 which owns the best offer. (4) Mangrove calls `makerExecute` at the address of Maker 1 with the volume required by Taker and a recap of the price agreed by Maker 1. (A) Maker 1 contract has now *gasreq* (of the offer) gas units to execute arbitrarily. (5) when `makerExecute` returns, Mangrove transfers the promised outbound tokens via a call to `transferFrom` on the ERC20 managing the outbound token. (6-9) Mangrove executes the same sequence on the offer of Maker 2. (10) At the end of the market order the sum of collected outbound assets are being sent to Taker as well as any left over inbound assets.

Proposition 1 (Compliance)**.** *A Maker is called by Mangrove on* `makerExecute` *if and only if:*

1. *The call occurs within a market order*
2. *The call has a gas limit set to the offer's* gasreq.
3. *The required volume is less or equal to the volume the Maker promised to send.*
4. *The price agreed by the taker is exactly the price at which the Maker promised to swap.*

Proposition 2 (Justice)**.** *Reverts that occur outside a call to* `makerExecute` *make the market order transaction revert.*

Arbitrage. When called by Mangrove on `makerExecute`, a Maker contract is given a certain amount of gas units to run (see (A) and (B) sections of Figure 1). This amount is specified by the Maker with the *gasreq* argument of the `newOffer` function. We wish to guarantee that the Maker cannot utilise this gas to *arbitrage* the trade: if Taker is willing to buy at price p and Maker made its offer at price $p' < p$, Maker could revert on the current offer (sending a fraction f of its provision to the Taker in compensation) and place a new offer on the list (with a reentrant call to `newOffer`) with a new price $p'' > p'$ and lower than the next best price on the offer list. This is guaranteed to be profitable for Maker if f is small enough in front of selling a volume V at price p'' instead of p'.

Proposition 3 (No Taker side leakage)**.** *During the execution of* `makerExecute`*, Mangrove's state does not give access to the limit price of the Taker's market order.*

Proposition 4 (No Maker side leakage)**.** *During the execution of* `makerExecute`*, offers from the same offer list are read and write locked.*

Market clogging This potential issue is in the category of griefing attacks, designed to harm protocol usability without immediate payoff for the attacker. The idea is to place offers so that any Takers' market orders end up reverting or unfilled (without the appropriate compensation).

Proposition 5 (Progress)**.** *For all offer list state, there exists a* productive *market order, i.e. that is either a non empty partial fill or returns a compensation for the Taker.*

Memory expansion attack. The reference EVM specification [3] states that allocating new elements to a smart contract's memory has a cost that depends on the size of the already allocated memory. Quoting the formula:

$$C_{mem}(a) := G_{memory} * a + \left\lfloor \frac{a^2}{512} \right\rfloor$$

where G_{memory} is a constant, and a is the number of memory words (of 32 bytes) allocated by the contract. Notice the cost of allocating a new word has a quadratic term beyond 22 words. During a market order, Mangrove is performing several calls to the Maker contracts whose offers are matched by the taker order. Each call is performed in a new context with a fresh memory, but when a call returns, one needs to make sure that the memory of Mangrove does not grow indefinitely. In particular care must be taken that a Maker cannot induce an uncontrolled growth of Mangrove's memory using polluted return data.

Proposition 6 (No memory leakage)**.** *The memory size of Mangrove is bounded by a constant at all time of a market order execution. This constant is independent of the offer list state.*

2 EVM-light

We proceed with the formal presentation of an EVM-light, which will serve as an operational semantics for smart contracts models we wish to analyse. This lightweight formalism makes several important simplifications with respect to the EVM. Those simplifications retain enough of the EVM semantics to be able to give value to the verification of the propositions listed in Section 1.4, while abstracting away the complex machinery of the EVM that would be irrelevant for the present study. In particular our EVM light has a very loose and flexible interpretation of values, disregarding size of elements stored in memory, which are normally constrained to EVM words of 32 bytes. So the present formalism cannot be used to study overflow issues. We also do not model stack allocation and consider contracts to be solely equipped with a memory heap on which to allocate variables. An additional important simplification is that we do not consider native token transfer and assume all payments to be made with ERC20 compliant custodian contracts.

2.1 Terminology

We consider a finite set of values *Vals* with a distinguished zero value $\mathbf{0}$. All common types (strings, numbers, booleans, addresses) are assumed to be transparently encoded to and from this value set (so for instance, any string is a value and any value is a string).

For convenience, we can talk about the set of strings $\mathbb{S}$, of addresses $\mathbb{A}$, of signed $\mathbb{Z}$ and unsigned integers $\mathbb{N}$, or of booleans $\mathbb{B}$. But we are really talking about the set of values.

Common operations such as $+$, $-$, comparison, etc are defined on values.

Variables We use variables, or names, as field names in the smart contract language, and also in the language semantics. There is a countable set of names *Vars*.

Memory Contracts use memory as a transient scratchpad. A *memory* or *mem* is a total mapping from names to values. The memory $(x \mapsto v, y \mapsto v')$ maps x to v, y to v', and maps every other name to $\mathbf{0}$. $\mathbf{0}_{mem}$ is the empty memory, it maps every name to $\mathbf{0}$.

Tuples We use tuples everywhere to structure definitions. A *tuple* is a partial mapping from names (to anything). $()$ is the empty tuple.

Example 1. *We will later introduce* contracts *as tuples of the form* $(abi \in \mathbb{S} \to \mathit{Bytecodes}, store \in \mathbb{N} \to \mathit{Vals})$. *This means that all contracts have domain* $\{abi, store\} \subseteq \mathit{Vars}$, *that they all map abi to strings-to-bytecode mappings, and that they all map store to integers-to-value mappings. Given concrete values* A *and* S, $C = (abi \mapsto A, store \mapsto D)$ *is the contract such that* $C(abi) = A$ *and* $C(store) = D$

When convenient the name of a tuple member is omitted and its position implicitly denotes the corresponding name.

Example 2. $C' = (A', D')$ *is the contract with domain* $\{abi, store\}$ *such that* $C'(abi) = A'$ *and* $C'(store) = D'$

Stores Contracts use their store (or storage) as long-term memory. A *store* is a total mapping from values (representing integers) to values. The store $(2 \mapsto v)$ maps 2 to v, and it maps every other integer to $\mathbf{0}$. $\mathbf{0}_{store}$ is the empty store, it maps every number to $\mathbf{0}$.

Arrays *Arrays* are partial mappings from integers (to anything) with no hole in the domain of definition. $[]$ is the empty array (aka the function defined nowhere). For any set X, the set of arrays with all elements in X is $\overrightarrow{X}$.

Example 3. $a = [3, 0, 1]$ *has domain* $\mathbf{3}$. $a(0) = 3$, $a(1) = 0$, $a(2) = 1$.

Update syntax A mapping can be defined as an update of an existing mapping. If a mapping m is defined on x then $m' = m\{x \mapsto v\}$ is equal to m, except that $m'(x) = v$. The notation is extended to more than one element.

Example 4. $C\{store \mapsto D'\}$ *is the contract* $(abi \mapsto A, store \mapsto D')$. $a\{1 \mapsto 10\}$ *is the array* $[3, 10, 1]$.

Nested update syntax If t is a tuple and $t.f$ is a mapping, then $t\{f\{\ldots\}\} = t\{f \mapsto t.f\{\ldots\}\}$.

Example 5. *If state is a tuple, state.address is a tuple, and state.address.store is a store, then :*

$$state\{address\{store\{3 \mapsto 7\}\}\}$$

is equal to state except that $state.address.store(3) = 7$.

2.2 Bytecodes

A *bytecode* B is a stack of *instructions* I, built using the grammar (for all n):

$$\begin{array}{rcll}
B & ::= & I; B \mid \mathbf{0} & \\
I & ::= & x := E \mid \texttt{sstore}(x_{\text{loc}}, E) & \text{Assignment} \\
 & & \mid \texttt{return}(x_0, \ldots, x_n) \mid \texttt{revert}(x_0, \ldots, x_n) & \text{Return and revert} \\
 & & \mid \texttt{if}(x_{\text{cond}}, B) \mid \texttt{while}(x_{\text{cond}}, B) & \text{Conditional} \\
 & & \mid \texttt{call}(x_{\text{to}}, x_{\text{fn}}, x_0, \ldots, x_n, x_{\text{gas}}) & \\
 & & \mid \texttt{dcall}(x_{\text{to}}, x_{\text{fn}}, x_0, \ldots, x_n, x_{\text{gas}}) & \text{Function calls} \\
E & ::= & op(x_0, \ldots, x_n) \mid \texttt{sload}(x_{\text{loc}}) \mid \texttt{hash}(x_0, \ldots, x_n) & \text{Expression} \\
 & & \mid \texttt{gasleft} \mid \texttt{this} \mid \texttt{sender} \mid \texttt{rdata}[i] \mid \texttt{cdata}[i] &
\end{array}$$

where $op : \mathit{Vals}^n \to \mathit{Vals}$ denote any operation on values (constant values are for $n = 0$), `hash`() gives access to a native hash function that maps values to integers, `gasleft`, `this` and `sender` give respectively access to the amount of gas unit left, the address of the execution runner and the address of the last caller of this execution. Each element `rdata`$[i]$ gives access to the *return data* values of the last call and `cdata`$[i]$ gives access to the *call data* values. By convention `rdata`$[0]$ contains the return status (a boolean) of the last function call[2].

We use *Bytecodes* to denote the set of bytecodes (words recognised by the above grammar).

2.3 Blockchain accounts and states.

Accounts. An *account* is a tuple of the form

$$\mathit{account} := (\mathit{abi} \in \mathbb{S} \to \mathit{Bytecodes}, \mathit{store} \in \mathbb{N} \to \mathit{Vals})$$

where *abi* is the account's application binary interface and *store* is the account's storage. *abi* maps strings (function names) to bytecodes and *store* maps positions to values in a persistent manner. We use *Accounts* to denote the set of possible accounts. A *contract* is an account A for which there exists at least a function symbol f such that $A.\mathit{abi}(f) \neq \mathbf{0}_{\mathit{Bytecodes}}$. The *effective interface* of an account A is a set of function symbols $I_A \subseteq \mathbb{S}$ such that for all $f \notin I_A$, $A.\mathit{abi}(f) = $ `revert("not defined")` and for all $f \in I_A$, $A.\mathit{abi}(f) \neq$ `revert("not defined")`. When modelling a smart contract we only mention its effective interface, and assume that all other function symbol map to the reverting bytecode on the reserved error message `"not defined"`. Note that externally owned accounts can be modelled as accounts whose ABI map all symbols to $\mathbf{0}_{\mathit{Bytecodes}}$ and whose store maps all position to $\mathbf{0}_{\mathit{Vals}}$.

State. A *(blockchain) state* is a mapping $\mathit{state} \in \mathbb{A} \to \mathit{Accounts}$ and the set of all possible states is *States*.

Example 6 (ERC20)**.** *Consider the contract* ERC20 *implementing the ERC20 standard for custodian contracts (we mention only the effective interface of the contract):*

$$\mathsf{ERC20}.\mathit{abi} := \begin{cases} \texttt{"name"} & \mapsto \texttt{return}(\text{“}\mathit{token\_name}\text{”}) \\ \texttt{"transfer"} & \mapsto B_{\mathit{transfer}} \\ \texttt{"transferFrom"} & \mapsto B_{\mathit{transferFrom}} \\ \texttt{"approve"} & \mapsto B_{\mathit{approve}} \\ \texttt{"allowance"} & \mapsto B_{\mathit{allowance}} \\ \texttt{"balanceOf"} & \mapsto B_{\mathit{balanceOf}} \end{cases}$$

and with store:

$$\mathsf{ERC20}.\mathit{store} := \begin{cases} s_0 & \mapsto \mathit{balance\_map} \in \mathbb{N} \\ s_1 & \mapsto \mathit{approval\_map} \in \mathbb{N} \\ s_2 & \mapsto \mathit{totalSupply} \in \mathbb{N} \\ \ldots & \end{cases}$$

with $s_i \in \mathbb{N}$ for $i \geq 0$.

[2]In the EVM, the return status is stored on top of the stack.

In Example 6, notice the constant function `name`() that returns a human readable name for the token managed by this contract without reading the contract's storage[3]. The *balance_map*, in the contract's storage, associates addresses to integers with $s_0[a] = n$ indicating that address a owns n tokens managed by this contract (see Section 4.1 for the interpretation of a mapping in storage). The *approval_map* associates fund owner addresses to spender addresses and their allowance, so $s_1[owner][spender] = n$ indicates that the address *spender* is allowed to transfer up to n tokens of *owner* to an arbitrary address, using the `transferFrom` function of the ERC20's interface. We will come back to the modelling of ERC20 contracts in Section 4.2.

3 Operational semantics

3.1 Execution steps and contexts

An *(execution) context* is a tuple of the form

$$\mathit{context} := (\mathit{calldata} \in \overrightarrow{\mathit{Vals}}, \mathit{this} \in \mathbb{A}, \mathit{sender} \in \mathbb{A})$$

where *calldata* is an array of values given by the caller, *this* is the address of the current execution runner, and *sender* is the address of the execution's caller.

An *(execution) step* is a tuple of the form

$$\mathit{step} := (\mathit{code} \in \mathit{Bytecodes}, \mathit{gas} \in \mathbb{Z}, \mathit{returndata} \in \overrightarrow{\mathit{Vals}}, \mathit{mem} \in \mathit{Vars} \to \mathit{Vals}, \mathit{state} \in \mathit{States})$$

The set of all steps is *Steps*.

3.2 Transaction initialisation

A *transaction* is given as a tuple of the form $tx := (\mathit{origin} \in \mathbb{A}, \mathit{to} \in \mathbb{A}, \mathit{fn} \in \mathbb{S}, \mathit{calldata} \in \overrightarrow{\mathit{Vals}}, \mathit{gas} \in \mathbb{N})$ where *origin* is the transaction originator, *to* is the address of the called contract, *fn* a function name and $\overrightarrow{c}$ its call data. The maximum amount of gas the transaction can spend is given by *gas*. We use *Transactions* to denote the set of transactions.

A transaction t executed on a blockchain *state* induces an initial execution state: $\mathit{context}_{tx} \vdash \mathit{step}_{\mathit{state},tx}$ where:

$$\begin{aligned} \mathit{context}_{tx} &:= (tx.\mathit{calldata}, tx.\mathit{to}, tx.\mathit{origin}) \\ \mathit{step}_{\mathit{state},tx} &:= (\mathit{state}(tx.\mathit{to}).\mathit{abi}(tx.\mathit{fn}), tx.\mathit{gas}, [], \mathbf{0}_{\mathit{mem}}, \mathit{state}) \end{aligned}$$

Example 7 (Example 6 continued)**.** *We consider a blockchain state that contains a deployed instance of our ERC20 contract:* $\mathit{state}_0 := (a_0 \mapsto \mathsf{ERC20})$ *and the following transaction:*

$$tx := (\mathit{origin} \mapsto o, \mathit{to} \mapsto a_0, \mathit{fn} \mapsto \texttt{"name"}, \mathit{calldata} \mapsto [], \mathit{gas} \mapsto G)$$

for some address $o \in \mathbb{A}$ and gas amount $G \in \mathbb{N}$. This transaction yields the initial execution state:

$$(\mathit{calldata} \mapsto [], \mathit{this} \mapsto a_0, \mathit{sender} \mapsto o) \vdash (\texttt{return}(\text{“}\mathit{token\_name}\text{”}), G, [], \mathbf{0}_{\mathit{mem}}, \mathit{state}_0)$$

3.3 Checking gas limit

An execution step is out of gas if it has less than 0 gas left. Derivation rules $\dashrightarrow$ below don't check gas, but the steps are only valid ($\to$) if the step has some gas left. In the derivation below, `"out of gas"` is a value that represents the string *out of gas*.

$$\frac{\mathit{context} \vdash \mathit{step} \dashrightarrow \mathit{step}' \quad \mathit{step}'.\mathit{gas} < 0}{\mathit{context} \vdash \mathit{step} \to \mathit{step}\,\{\mathit{code} \mapsto \texttt{revert("out of gas")}\}} \tag{1}$$

$$\frac{\mathit{context} \vdash \mathit{step} \dashrightarrow \mathit{step}' \quad \mathit{step}'.\mathit{gas} \geq 0}{\mathit{context} \vdash \mathit{step} \to \mathit{step}'} \tag{2}$$

[3] This models immutable constants of the Solidity language

3.4 Final steps

A *step* that does not rewrite to any step through $\rightarrow$ is *final*. We denote it by *step* $\not\rightarrow$.

Returning steps. A *step returns* an array $r \in \overrightarrow{Vals}$ when either:

- $step.code = \texttt{return}(x_1, \ldots, x_n); B$, for some variables x_i s and code B, in which case:

$$r = [true, step.mem(x_1), \ldots, step.mem(x_n)]$$

- $step.code = \mathbf{0}_{\texttt{B}}$, in which case $r = [true]$.

Property 1. *Any returning step is final.*

Reverting steps. Any non-returning final *step reverts* an array $r \in \overrightarrow{Vals}$ with:

- $r = [false, step.mem(x_1), \ldots, step.mem(x_n)]$ when $step.code = \texttt{revert}(x_1, \ldots, x_n); B$, for some variables x_is and code B.
- $r = [false]$ otherwise.

3.5 Function calls

Call step. A *step calls* $(a \in \mathbb{A}, f \in \mathbb{S}, B \in Bytecodes, c \in \overrightarrow{Vals}, g \in \mathbb{N})$ *with continuation* $B' \in Bytecodes$ when

$$\begin{array}{lcl} step.code & = & (\texttt{d})\texttt{call}(x_{\text{to}}, x_{\text{fn}}, x_1, \ldots, x_n, x_{\text{gas}}); B' \\ a & = & step.mem(x_{\text{to}}) \\ f & = & step.mem(x_{\text{fn}}) \\ B & = & state(a).abi(f) \\ c & = & [step.mem(x_1), \ldots, step.mem(x_n)] \\ g & = & step.mem(x_{\text{gas}}) \end{array}$$

We say that the above call step *delegates* if the opcode is dcall and not call.

Derivation of a function call.

$$\frac{\begin{array}{l} step \text{ calls } (a, f, B, c, g) \text{ with continuation } B' \\ context' = \begin{cases} context\,\{calldata \mapsto c\} & \text{if } step \text{ delegates} \\ (calldata \mapsto c, this \mapsto a, sender \mapsto context.this) & \text{otherwise} \end{cases} \\ context' \vdash (B, g, [], \mathbf{0}_{mem}, step.state) \rightarrow^* step' \not\rightarrow \\ g' = step.gas - \delta_{\mathsf{call}} + (step'.gas - g) \\ state' = \begin{cases} step'.state \text{ if } step' \text{ returns } r \\ step.state \text{ if } step \text{ reverts } r \end{cases} \end{array}}{context \vdash step \dashrightarrow (B', g', r, step.mem, state')}\ (\texttt{d})\texttt{call}$$

Property 2. *Assume a step that calls* (a, f, B, c, g) *with continuation* B'. *The following properties hold:*

- *If* $step.state(a)$ *is an external account, then* $B = \mathbf{0}_{Bytecodes}$.
- *If* $step.state(a)$ *is a contract, but* f *is not an element of its interface, then* $B = \texttt{revert}()$.

3.6 Storage and memory allocation

Evaluation function. We first define the eval function that maps an expression to a value and an evaluation gas cost:

$$\begin{array}{lcll}
\mathsf{eval}(\mathit{context}, \mathit{gas}, r, \mathit{mem}, \mathit{state}, \mathit{op}(x)) & = & (\mathit{op}(s.\mathit{mem}(x)), & \delta_{\mathsf{mr}}) \\
\mathsf{eval}(\mathit{context}, \mathit{gas}, r, \mathit{mem}, \mathit{state}, \mathtt{sload}(x_{\mathbb{N}})) & = & (s.\mathit{state}(x_{\mathbb{N}}), & \delta_{\mathsf{sr}}) \\
\mathsf{eval}(\mathit{context}, \mathit{gas}, r, \mathit{mem}, \mathit{state}, \mathtt{gasleft}) & = & (s.\mathit{gas}, & \delta_{\mathsf{env}}) \\
\mathsf{eval}(\mathit{context}, \mathit{gas}, r, \mathit{mem}, \mathit{state}, \mathtt{this}) & = & (\mathit{context}.\mathit{this}, & \delta_{\mathsf{env}}) \\
\mathsf{eval}(\mathit{context}, \mathit{gas}, r, \mathit{mem}, \mathit{state}, \mathtt{sender}) & = & (\mathit{context}.\mathit{sender}, & \delta_{\mathsf{env}}) \\
\mathsf{eval}(\mathit{context}, \mathit{gas}, r, \mathit{mem}, \mathit{state}, \mathtt{cdata}[i]) & = & (s.\mathit{calldata}[i], & \delta_{\mathsf{dcd}}) \\
\mathsf{eval}(\mathit{context}, \mathit{gas}, r, \mathit{mem}, \mathit{state}, \mathtt{rdata}[i]) & = & (s.\mathit{returndata}[i], & \delta_{\mathsf{drd}}) \\
\mathsf{eval}(\mathit{context}, \mathit{gas}, r, \mathit{mem}, \mathit{state}, \mathtt{hash}(x_0, \dots, x_n)) & = & (\mathit{keccak}_{256}(v_0 :: \cdots :: v_n), & \delta_{\mathsf{hsh}} * n)
\end{array}$$

with $v_i = \mathit{mem}(x_i)$, and $v :: v'$ denoting value concatenation.

Storage and memory access derivations.

$$\frac{\mathsf{eval}(\mathit{context}, \mathit{gas}, r, \mathit{mem}, \mathit{state}, E) = (v, \delta)}{\mathit{context} \vdash (x := E; B, \mathit{gas}, r, \mathit{mem}, \mathit{state}) \dashrightarrow (B, \mathit{gas} - \delta - \delta_{\mathsf{mw}}, r, \mathit{mem}\{x \mapsto v\}, \mathit{state})}\ \mathtt{mstore}$$

$$\frac{\mathsf{eval}(\mathit{context}, \mathit{gas}, r, \mathit{mem}, \mathit{state}, E) = (v, \delta) \quad \mathit{state}' = \mathit{state}\{\mathit{context}.\mathit{this}\{\mathit{store}\{\mathit{mem}(x_{\text{pos}}) \mapsto v\}\}\}}{\mathit{context} \vdash (\mathtt{sstore}(x_{\text{pos}}, E); B, \mathit{gas}, r, \mathit{mem}, \mathit{state}) \dashrightarrow (B, \mathit{gas} - \delta - \delta_{\mathsf{sw}}, r, \mathit{mem}, \mathit{state}')}\ \mathtt{sstore}$$

3.7 Conditionals and loops

$$\frac{\mathit{step}.\mathit{code} = \mathtt{if}(x_{\text{cond}}, B'); B \quad B'' = \begin{cases} B & \text{if } s.\mathit{mem}(x_{\text{cond}}) = \mathbf{0}_{\mathbb{B}} \\ B'; B & \text{otherwise} \end{cases}}{\mathit{context} \vdash s \dashrightarrow s\{\mathit{gas} \mapsto g - \delta_{\mathsf{mr}}, \mathit{code} \mapsto B''\}}\ \mathtt{if}$$

$$\frac{\mathit{step}.\mathit{code} = \mathtt{if}(x_{\text{cond}}, B'); B \quad B'' = \begin{cases} B & \text{if } s.\mathit{mem}(x_{\text{cond}}) = \mathbf{0}_{\mathbb{B}} \\ B'; \mathtt{while}(x_{\text{cond}}, B'); B & \text{otherwise} \end{cases}}{\mathit{context} \vdash \mathit{step} \dashrightarrow \mathit{step}\{\mathit{gas} \mapsto g - \delta_{\mathsf{mr}}, \mathit{code} \mapsto B''\}}\ \mathtt{while}$$

Figure 2: Conditional and while loop.

4 Modelling smart contracts

4.1 Solidity light

We can define some basic solidity constructs as macros (introduced variables are fresh) that we will use to model smart contracts. First, opcodes that require variable arguments can be relaxed to expect expressions, letting the mstore rule of Section 3.6 evaluate statements of the form $x := E$:

$$\mathsf{while}/\mathtt{if}(E, B) \equiv x_{\text{cond}} := E; \mathsf{while}/\mathtt{if}(x_{\text{cond}}, B)$$

$$\mathsf{revert}/\mathtt{return}(E_0, \ldots, E_n) \equiv x_0 := E_0; \ldots; x_n := E_n; \mathsf{revert}/\mathtt{return}(x_0, \ldots, x_n)$$

$$(\mathsf{d})\mathtt{call}(E_{\text{to}}, E_{\text{fn}}, E_0, \ldots, E_n, E_{\text{gas}}) := \begin{cases} x_{\text{to}} := E_{\text{to}}; x_{\text{fn}} := E_{\text{fn}}; \\ x_0 := E_0; \ldots; x_n := E_n; \\ x_{\text{gas}} := E_{\text{gas}} \\ (\mathsf{d})\mathtt{call}(x_{\text{to}}, x_{\text{fn}}, x_0, \ldots, x_n, x_{\text{gas}}) \end{cases}$$

$$\mathtt{hash}(E_0, \ldots, E_n) \equiv x_0 := E_0; \ldots; x_n := E_n; \mathtt{hash}(x_0, \ldots, x_n)$$

We can then model core solidity primitives:

$$\mathtt{require}(E_{\text{cond}}, E_{\text{reason}}) \equiv \mathtt{if}(\neg E_{\text{cond}}, \mathtt{revert}(E_{\text{reason}}))$$

$$a\,.\,f\{\mathtt{gas:}E_{\text{gas}}\}(E_0, \ldots, E_n) \equiv \begin{cases} \mathtt{call}(a, f, E_0, \ldots, E_n, E_{\text{gas}}); \\ \mathtt{require}(\mathtt{rdata}[0], \mathtt{rdata}[1]) \end{cases}$$

$$\begin{array}{l} \mathtt{try}\ a\,.\,f\{\mathtt{gas:}E_{\text{gas}}\}(E_0, \ldots, E_n) \\ \mathtt{returns}(x_0, \ldots, x_m)\{B\ \} \\ \mathtt{catch}\ \{B'\} \end{array} \equiv \begin{cases} \mathtt{call}(a, f, E_0, \ldots, E_n, E_{\text{gas}}); \\ \mathtt{if}(\mathtt{rdata}[0], (x_i := \mathtt{rdata}[i+1])_{i \leq |\mathtt{rdata}|}); B) \\ \mathtt{if}(\neg\mathtt{rdata}[0], B') \end{cases}$$

$$z := E_{\text{map}}[E_0] \ldots [E_n] \equiv z_{\text{loc}} := \mathtt{hash}(E_{\text{map}}, E_0, \ldots, E_n); z := \mathtt{sload}(z_{\text{loc}})$$

$$E_{\text{map}}[E_0] \ldots [E_n] := E \equiv z_{\text{loc}} := \mathtt{hash}(E_{\text{map}}, E_0, \ldots, E_n); \mathtt{sstore}(z_{\text{loc}}, E)$$

4.2 A simple custodian contract

References

[1] Mangrove. The “offer-is-code” approach to decentralised exchanges. Technical report, Mangrove whitepaper, 2021.

[2] Fabian Vogelsteller. https://eips.ethereum.org/EIPS/eip-20.

[3] Gavin Wood. Ethereum: a secure decentralised generalised transaction ledger. Technical report, Ethereum and Parity, Apr 2024.