A blockchain calculus

A. Husson, J. Krivine

1 Terminology

1.1 Terminology

Values and variables We consider a finite set of values *Vals* with a distinguished zero value **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 unsigned integers $\mathbb{U}$, 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 **0**. $\mathbf{0}_{mem}$ is the empty memory, it maps every name to **0**.

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

Example We will later introduce *contracts* as tuples of the form $(ABI \in \mathbb{S} \to \textit{Bytecodes}, \textit{store} \in \mathbb{N} \to \textit{Vals})$. This means that all contracts have domain $\{ABI, store\} \subseteq \textit{Vars}$, 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 $C' = (A', D')$ 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 **0**. $\mathbf{0}_{store}$ is the empty store, it maps every number to **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 $\vec{X}$.

Examples $a = [3, 0, 1]$ has domain **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 $C\{store \mapsto D'\}$ 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 $t\{f\{\dots\}\} = t\{f \mapsto t.f\{\dots\}\}$.

Example 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 $state.address.store(3) = 7$.

1.2 Bytecodes

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

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

where $op : \mathit{Vals}^n \to \mathit{Vals}$ denote any operation on values (constant values are for $n = 0$), $\mathtt{hash}()$ gives access to a native hash function that maps values to integers, $\mathtt{gasleft}$, $\mathtt{this}$ and $\mathtt{sender}$ 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 $\mathtt{rdata}[i]$ gives access to the *return data* values of the last call and $\mathtt{cdata}[i]$ gives access to the *call data* values.

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

Example We can define some basic solidity constructs as macros (introduced variables are fresh):

$$\mathtt{if}(E, B) \quad := \quad x_{\mathrm{cond}} := E; \mathtt{if}(x_{\mathrm{cond}}, B)$$

$$\ldots \quad := \quad \ldots$$

$$\mathtt{dcall}(E_{\mathrm{to}}, E_{\mathrm{fn}}, E_0, \ldots, E_n, E_{\mathrm{gas}}) \quad := \quad \begin{cases} x_{\mathrm{to}} := E_{\mathrm{to}}; x_{\mathrm{fn}} := E_{\mathrm{fn}}; \\ x_0 := E_0; \ldots; x_n := E_n; \\ x_{\mathrm{gas}} := E_{\mathrm{gas}} \\ \mathtt{dcall}(x_{\mathrm{to}}, x_{\mathrm{fn}}, x_0, \ldots, x_n, x_{\mathrm{gas}}) \end{cases}$$

$$\mathtt{require}(E_{\mathrm{cond}}, E_{\mathrm{reason}}) \quad := \quad \mathtt{if}(\neg E_{\mathrm{cond}}, \mathtt{revert}(E_{\mathrm{reason}}))$$

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

$$\begin{array}{l} \mathtt{try}\ a.f\{\mathtt{gas}\!:\!E_{\mathrm{gas}}\}(E_0, \ldots, E_n) \\ \mathtt{returns}(x_0, \ldots, x_m) \\ \{B\}\ \mathtt{catch}\ \{B'\} \end{array} \quad := \quad \begin{cases} \mathtt{call}(a, f, \vec{v}, g); success_{\mathbb{B}} := \mathtt{rdata}_0 \\ \mathtt{if}(success_{\mathbb{B}}, (y_i := \mathtt{rdata}_{i+1})_{i \leq |\mathtt{rdata}|}); B) \\ \mathtt{if}(\neg success_{\mathbb{B}}, B') \end{cases}$$

$$z := x_{\mathbb{N}}[y] \quad := \quad y'_{\mathbb{N}} := \mathtt{hash}(y) + x_{\mathbb{N}}; z := \mathtt{sload}(z_{\mathbb{N}})$$

1.3 Smart contracts

A smart contract Ctr is a tuple of the form

$$\mathsf{Ctr} := (ABI \in \mathbb{S} \to Bytecodes, store \in \mathbb{N} \to Vals)$$

where ABI is the contract's *(application binary) interface*, mapping a function name to its bytecode. The contract storage maps positions to values in a persistent manner. We use *Contracts* to denote the set of possible smart contracts.

A *(blockchain) state* is a mapping $state \in \mathbb{A} \to Contracts$. The set of all possible states is *States*.

2 Operational semantics

2.1 Execution steps and environment

An *(execution) context* is a tuple of the form $\mathsf{Ctx} := (calldata \in \vec{Vals}, this \in \mathbb{A}, sender \in \mathbb{A})$, 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 $\mathsf{Step} := (code \in Bytecodes, gas \in \mathbb{U}, returndata \in \vec{Vals}, mem \in Vars \to Vals, state \in States)$. The set of all steps is *Steps*.

2.2 Transaction initialization

A *transaction t* is given as a tuple of the form $\text{tx} := (origin \in \mathbb{A}, to \in \mathbb{A}, fn \in \mathbb{S}, calldata \in \vec{Vals}, gas \in \mathbb{U})$ where *origin* is the transaction originator, *to* is the address of the called contract, *fn* a function name and $\vec{c}$ 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 tx executed on a blockchain *state* induces an initial context Ctx_{tx} and an initial step Step_{tx}:

$$\text{Ctx}(\text{tx}) \quad := \quad (\text{tx}.origin, \text{tx}.to, \text{tx}.calldata)$$

$$\text{Step}(\text{tx}) \quad := \quad state \mapsto (state(\text{tx}.to).ABI(\text{tx}.fn), \text{tx}.gas, [], \mathbf{0}_{mem}, state)$$

2.3 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 ($\rightarrow$) if the step has some gas left. In the derivation below, `"out of gas"` is a value that represents the string *out of gas*.

$$\frac{\text{Ctx} \vdash \text{Step} \dashrightarrow \text{Step}' \quad \text{Step}'.gas < 0}{\text{Ctx} \vdash \text{Step} \rightarrow \text{Step}\{B \mapsto \text{revert}(\text{"out of gas"}); \mathbf{0}_{\text{B}}\}} \tag{1}$$

$$\frac{\text{Ctx} \vdash \text{Step} \dashrightarrow \text{Step}' \quad \text{Step}'.gas \geq 0}{\text{Ctx} \vdash \text{Step} \rightarrow \text{Step}'} \tag{2}$$

2.4 Function calls derivations

2.4.1 Final steps

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

Returning steps   A step s *returns* an array r when either:

- $r = [true, s.mem(x_1), \ldots, s.mem(x_n)]$ if $s.code = \texttt{return}(x_1, \ldots, x_n); B$, for some x_i s and B, and

- $r = [true]$ if $s.code = \mathbf{0}_{\mathsf{B}}$

   Exercise   Show that any returning step is final (hint: check derivation rules).

Reverting steps   Any non-returning final step s reverts r, where

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

- $r = [false]$ otherwise

2.4.2 Context nesting

We say that $\mathsf{Ctx} \vdash s$ *creates* $\mathsf{Ctx}' \vdash (B', G')|B$ when

$$
\left\{
\begin{array}{rcl}
s.code & = & \texttt{call}(x_{\mathbb{A}}, x_{\mathbb{S}}, x_1, \ldots, x_n, x_{\mathbb{N}}); B \\
\mathsf{Ctx}' & = & (calldata \mapsto c, this \mapsto a, sender \mapsto \mathsf{Ctx}.this)
\end{array}
\right.
$$

or

$$
\left\{
\begin{array}{rcl}
s.code & = & \texttt{dcall}(x_{\mathbb{A}}, x_{\mathbb{S}}, x_1, \ldots, x_n, x_{\mathbb{N}}); B \\
\mathsf{Ctx}' & = & \mathsf{Ctx}\{calldata \mapsto c\},
\end{array}
\right.
$$

and
$$
\left\{
\begin{array}{rcl}
s.mem(x_{\mathbb{A}}) & = & a \\
s.mem(x_{\mathbb{S}}) & = & f \\
s.mem(x_i) & = & c[i] \quad \forall i \leq n \\
s.mem(x_{\mathbb{N}}) & = & G' \\
B' & = & s.state(a).ABI(f) \\
s.gas & \geq & G' + \delta_{\mathsf{call}}
\end{array}
\right.
$$

for some $x_{\mathbb{A}}, \ldots, x_{\mathbb{N}}$, c,a, and f. This definition applies to all (context,step) pairs that create a new call frame, whether through a call or a delegatecall.

Ctx' is the call context. For a call, it changes the values of *this* and *sender*. For a delegatecall, it keeps those of Ctx.

B' is the bytecode of the new call frame and corresponds to the code at the given function (f) name at the given address (a).

G' is the gas given for the call.

B is the continuation bytecode after the call has ended (i.e. the rest of the bytecode after s's call instruction).

2.4.3 Derivation of a function call

$$\begin{array}{c}
\mathsf{Ctx} \vdash s \text{ creates } \mathsf{Ctx'} \vdash (B', G')|B \\
\mathsf{Ctx'} \quad \vdash \quad (B', G, [], \mathbf{0}_{mem}, s.state) \to^* t \not\to \\
state \quad := \quad \begin{cases} t.state \text{ if } t \text{ returns } r \\ s.state \text{ if } t \text{ reverts } r \end{cases} \\
\dfrac{gas \quad := \quad s.gas - \delta_{\mathsf{call}} + t.gas - G'}{\mathsf{Ctx} \vdash s \dashrightarrow (B, gas, r, s.mem, state)} \; \texttt{xcall}
\end{array}$$

2.5 Storage and memory allocation

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

$$\begin{array}{lcl}
\mathsf{eval}(\mathsf{Ctx}, gas, r, mem, state, op(x)) & = & (op(s.mem(x)), \quad \delta_{\mathsf{mr}}) \\
\mathsf{eval}(\mathsf{Ctx}, gas, r, mem, state, \texttt{sload}(x_{\mathbb{N}})) & = & (s.state(x_{\mathbb{N}}), \quad \delta_{\mathsf{sr}}) \\
\mathsf{eval}(\mathsf{Ctx}, gas, r, mem, state, \texttt{gasleft}) & = & (s.gas, \quad \delta_{\mathsf{env}}) \\
\mathsf{eval}(\mathsf{Ctx}, gas, r, mem, state, \texttt{this}) & = & (\mathsf{Ctx}.this, \quad \delta_{\mathsf{env}}) \\
\mathsf{eval}(\mathsf{Ctx}, gas, r, mem, state, \texttt{sender}) & = & (\mathsf{Ctx}.sender, \quad \delta_{\mathsf{env}}) \\
\mathsf{eval}(\mathsf{Ctx}, gas, r, mem, state, \texttt{cdata}[i]) & = & (s.calldata[i], \quad \delta_{\mathsf{dcd}}) \\
\mathsf{eval}(\mathsf{Ctx}, gas, r, mem, state, \texttt{rdata}[i]) & = & (s.returndata[i], \quad \delta_{\mathsf{drd}})
\end{array}$$

$$\dfrac{\mathsf{eval}(\mathsf{Ctx}, gas, r, mem, state, E) = (v, \delta)}{\mathsf{Ctx} \vdash (x := E; B, gas, r, mem, state) \dashrightarrow (B, gas - \delta - \delta_{\mathsf{mw}}, r, mem\{x \mapsto v\}, state)} \; \texttt{mstore} \quad \dfrac{\mathsf{eval}(\mathsf{Ctx}}{\mathsf{Ctx} \vdash}$$

Figure 1: Allocation rules.

2.6 Conditionals and loops

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

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

Figure 2: Conditional and while loop.

2.7 Abstract custodian contracts