# 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 $(ABI \in \mathbb{S} \to Bytecodes, store \in \mathbb{N} \to Vals)$. This means that all contracts have domain $\{ABI, store\} \subseteq Vars$, 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 $n$):

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

where $op : Vals^n \to Vals$ 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.

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):

$$
\begin{array}{lcl}
\texttt{if}(E, B) & := & x_{\text{cond}} := E; \texttt{if}(x_{\text{cond}}, B) \\
\ldots & := & \ldots \\
\texttt{dcall}(E_{\text{to}}, E_{\text{fn}}, E_0, \ldots, E_n, E_{\text{gas}}) & := & \left\{ \begin{array}{l} x_{\text{to}} := E_{\text{to}}; x_{\text{fn}} := E_{\text{fn}}; \\ x_0 := E_0; \ldots; x_n := E_n; \\ x_{\text{gas}} := E_{\text{gas}} \\ \texttt{dcall}(x_{\text{to}}, x_{\text{fn}}, x_0, \ldots, x_n, x_{\text{gas}}) \end{array} \right. \\
\texttt{require}(E_{\text{cond}}, E_{\text{reason}}) & := & \texttt{if}(\neg E_{\text{cond}}, \texttt{revert}(E_{\text{reason}})) \\
a\,.\,f\{\texttt{gas:}E_{\text{gas}}\}(E_0, \ldots, E_n) & := & \left\{ \begin{array}{l} \texttt{call}(a, f, E_0, \ldots, E_n, E_{\text{gas}}); \\ \texttt{require}(\texttt{rdata}[0], \texttt{rdata}[1]) \end{array} \right. \\
\begin{array}{l} \texttt{try } a\,.\,f\{\texttt{gas:}E_{\text{gas}}\}(E_0, \ldots, E_n) \\ \texttt{returns}(x_0, \ldots, x_m) \\ \{B\} \texttt{ catch } \{B'\} \end{array} & := & \left\{ \begin{array}{l} \texttt{call}(a, f, \vec{v}, g); \mathit{success}_{\mathbb{B}} := \texttt{rdata}_0 \\ \texttt{if}(\mathit{success}_{\mathbb{B}}, (y_i := \texttt{rdata}_{i+1})_{i \leq |\texttt{rdata}|}); B) \\ \texttt{if}(\neg \mathit{success}_{\mathbb{B}}, B') \end{array} \right. \\
z := x_{\mathbb{N}}[y] & := & y'_{\mathbb{N}} := \texttt{hash}(y) + x_{\mathbb{N}}; z := \texttt{sload}(z_{\mathbb{N}})
\end{array}
$$

## 1.3 Smart contracts

A smart contract $\mathsf{Ctr}$ is a tuple of the form

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

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

## 2.2 Transaction initialization

A *transaction* $t$ is given as a tuple of the form $\mathsf{tx} := (\mathit{origin} \in \mathbb{A}, \mathit{to} \in \mathbb{A}, \mathit{fn} \in \mathbb{S}, \mathit{calldata} \in \vec{\mathit{Vals}}, \mathit{gas} \in \mathbb{U})$ 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 $\mathsf{tx}$ executed on a blockchain *state* induces an initial context $\mathsf{Ctx}_{\mathsf{tx}}$ and an initial step $\mathsf{Step}_{\mathsf{tx}}$:

$$\begin{aligned} \mathsf{Ctx}(\mathsf{tx}) &:= (\mathsf{tx}.\mathit{origin}, \mathsf{tx}.\mathit{to}, \mathsf{tx}.\mathit{calldata}) \\ \mathsf{Step}(\mathsf{tx}) &:= \mathit{state} \mapsto (\mathit{state}(\mathsf{tx}.\mathit{to}).\mathit{ABI}(\mathsf{tx}.\mathit{fn}), \mathsf{tx}.\mathit{gas}, [], \mathbf{0}_{\mathit{mem}}, \mathit{state}) \end{aligned}$$

## 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*.

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

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

## 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 $s.code = \mathbf{0}_{\mathtt{B}}$

**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 $x_i$s 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

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

or

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

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

### 2.4.3 Derivation of a function call

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

## 2.5 Storage and memory allocation

We first define the $\mathsf{eval}$ function that maps an expression to value and a gas cost:

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

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

Figure 1: Allocation rules.

## 2.6 Conditionals and loops

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

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

Figure 2: Conditional and while loop.

## 2.7 Abstract custodian contracts