The Impact VM

> INFO
>
> Impact is still under active revision. Expect its attributes, including storage-related costs, to change.
>
> Currently, users cannot write Impact manually; this feature may be added in the future.

On-chain parts of programs are written in *Impact*, our on-chain VM language. You should not need to worry about the details of impact when writing contracts; however, you may see it appear when inspecting transactions and contract outputs.

Impact is a stack-based, non-Turing-complete state manipulation language. A contract is executed on a stack containing three things:

- a 'context' object describing context related to the containing transaction
- an 'effects' object gathering actions performed by the contract during the execution
- the contract's current state.

Program execution proceeds linearly, with no operations being able to decrease the program counter and every operation being bounded in the time it takes. Program execution has an attached cost, which may be bounded by a 'gas' limit. Programs can eit abort, invalidating this (part of) a transaction, or succeed, in which case they must leave a stack in the same shape as they star resulting effects must match the transcript's declared effects, and the contract state must be marked as storable, in which case adopted as the updated state.

Transcripts

Execution transcripts consist of:

- a declared gas bound, used to derive the fees for this call
- a declared effects object, used to bind the contract's semantics to that of other parts
- the program to execute.

Values

The Impact stack machine operates on the following state values:

- `null`
- `<x: y>`, a field-aligned binary cell
- `Map { k1: v1, k2: v2, ... }`, a map from field-aligned binary values to state values
- `Array(n) [ v0, v1, ... ]`, an array of `0 < n < 16` state values
- `MerkleTree(d) { k0: v1, k2: v2, ... }`, a sparse, fixed-depth `1 <= d <= 32` Merkle tree, with the slots `k0`, `k2`, ..., containing the leaf hashes `v1`, `v2`, ... (typically represented as hex strings).

Field-aligned binary

The basic data types used in Impact are 'field-aligned binary' (FAB) values. These values can store complex data structures in a binary representation while keeping the information necessary to encode them as field elements in any prime field.

Aligned values consist of a sequence of aligned *atoms*, each of which consists of a byte string and an alignment atom, where alignment atoms are one of:

- `f`, indicating a field alignment: the atom will be interpreted as a little-endian representation of a field element.

Feedback

- `c`, indicating a compression alignment: the atom will be interpreted as a field element derived by hashing its value.
- `b`*n*, indicating an `n`-byte alignment: the atom will be interpreted as a sequence of field elements depending on the prime field and curve to compactly encode `n` bytes.

Programs

A program is a sequence of operations, consisting of an opcode, potentially followed by a number of arguments depending on the specific opcode. Programs can be run in two modes: *evaluating* and *verifying*. In verifying mode, `popeq[c]` arguments are enforced for equality, while in evaluating mode, the results are gathered instead.

Each `Op` has a fixed effect on the stack, which will be written as `-{a, b} +{c, d}`: consuming items `a` and `b` being at the top of the stack (with `a` above `b`), and replacing them with `c` and `d` (with `d` above `c`). The number of values here is just an example. State values are *immutable* from the perspective of programs: a value on the stack cannot be changed, but it can be replaced with a modified version of the same value. We write `[a]` to refer to the value stored in the cell `a`. Due to the ubiquity of it, we write 'sets `[a] := ...`' for 'create `a` as a new cell containing `...`'. We prefix an output value with a `'` to indicate this is a *weak* value, kept solely in-memory, and not written to disk, and an input value with `'` to indicate it *may* be a weak value. We use `"` and `†` to indicate that an input *may* be a weak value, and *iff* it is, the correspondingly marked output will be a weak value.

Where arguments are used, we use `State` for a state value, `u21` for a 21-bit unsigned integer, and `path(n)` for a sequence of *either* field-aligned binary values, *or* the symbol `stack`, indicating keys to use in indexing, either directly, or to use stack values instead.

Name	Opcode	Stack	Arguments	Cost (unscaled)	Description
`noop`	`00`	`-{}` `+{}`	`n: u21`	`n`	nothing

Name	Opcode	Stack	Arguments	Cost (unscaled)	Description
`lt`	`01`	`-{'a, 'b}` `+{c}`	-	`1`	sets `[c] := [a] < [b]`
`eq`	`02`	`-{'a, 'b}` `+{c}`	-	`1`	sets `[c] := [a] == [b]`
`type`	`03`	`-{'a}` `+{b}`	-	`1`	sets `[b] := typeof(a)`
`size`	`04`	`-{'a}` `+{b}`	-	`1`	sets `[b] := size(a)`
`new`	`05`	`-{'a}` `+{b}`	-	`1`	sets `[b] := new [a]`
`and`	`06`	`-{'a, 'b}` `+{c}`	-	`1`	sets `[c] := [a] & [b]`
`or`	`07`	`-{'a, 'b}` `+{c}`	-	`1`	sets `[c] := [a]
`neg`	`08`	`-{'a}` `+{b}`	-	`1`	sets `[b] := ![a]`

Feedback

Name	Opcode	Stack	Arguments	Cost (unscaled)	Description
`log`	`09`	`-{'a}` `+{}`	-	`1`	outputs `a` as an event
`root`	`0a`	`-{'a}` `+{b}`	-	`1`	sets `[b] := root(a)`
`pop`	`0b`	`-{'a}` `+{}`	-	`1`	removes `a`
`popeq`	`0c`	`-{'a}` `+{}`	`a: State` only when validating	`	a
`popeqc`	`0d`	`-{'a}` `+{}`	`a: State` only when validating	`	a
`addi`	`0e`	`-{'a}` `+{b}`	`c: State`	`1`	sets `[b] := [a] + c`, where addition is defined below
`subi`	`0f`	`-{'a}` `+{b}`	`c: State`	`1`	sets `[b] := [a] - c`, where subtraction is defined below

Feedback

Name	Opcode	Stack	Arguments	Cost (unscaled)	Description
`push`	`10`	`-{}` `+{'a}`	`a: State`	`	a
`pushs`	`11`	`-{}` `+{a}`	`a: State`	`	a
`branch`	`12`	`-{'a}` `+{}`	`n: u21`	`1`	if `a` is non-empty, skip `n` operations.
`jmp`	`13`	`-{}` `+{}`	`n: u21`	`1`	skip `n` operations.
`add`	`14`	`-{'a, 'b}` `+{c}`	-	`1`	sets `[c] := [a] + [b]`
`sub`	`15`	`-{'a, 'b}` `+{c}`	-	`1`	sets `[c] := [b] - [a]`
`concat`	`16`	`-{'a, 'b}` `+{c}`	`n: u21`	`1`	sets `[c] = [b] ++ [a]`, if`
`concatc`	`17`	`-{'a, 'b}` `+{c}`	`n: u21`	`1`	as `concat`, but `a` and `b` must already be in-memory

Feedback

Name	Opcode	Stack	Arguments	Cost (unscaled)	Description
`member`	`18`	`-{'a, 'b}` `+{c}`	-	`size(b)`	sets `[c] := has_key(b, a)`
`rem`	`19`	`-{a, "b}` `+{"c}`	-	`size(b)`	sets `c := rem(b, a, false)`
`remc`	`1a`	`-{a, "b}` `+{"c}`	-	`size(b)`	sets `c := rem(b, a, true)`
`dup`	`3n`	`-{x*, "a}` `+{"a, x*, "a}`	-	`1`	duplicates `a`, where `x*` are `n` stack items
`swap`	`4n`	`-{"a, x*, †b}` `+{†b, x*, "a}`	-	`1`	swaps two stack items, with `n` items `x*` between them
`idx`	`5n`	`-{k*, "a}` `+{"b}`	`c: path(n)`	`	c
`idxc`	`6n`	`-{k*, "a}` `+{"b}`	`c: path(n)`	`	c

Name	Opcode	Stack	Arguments	Cost (unscaled)	Description
`idxp`	`7n`	`-{k*, "a}` `+{"b, pth*}`	`c: path(n)`	`	c
`idxpc`	`8n`	`-{k*, "a}` `+{"b, pth*}`	`c: path(n)`	`	c
`ins`	`9n`	`-{"a, pth*}` `+{†b}`	-	`sum size(x_i)`	where `pth*` is `{key_{n+1}, x_{n+1}, ..., key_1, x_1}` set `x'_{n+2} = a`, `x'_j = ins(x_j, key_j, cached, x'_{j+1})`, `b = x'_1`. `†` is the weakest modifier of `a` and `x_j`s, and `cached` set to `false`
`insc`	`an`	`-{"a, pth*}` `+{†b}`	-	`sum size(x_i)`	as `ins`, but with `cached` set to `true`
`ckpt`	`ff`	`-{}` `+{}`		`1`	denotes boundary between internally atomic program segments. Should not be crossed by jumps.

In the description above, the following short-hand notations are used. Where not specified, result values are placed in a `Cell` and encoded as FAB values.

Feedback

- `a + b`, `a - b`, or `a < b` (collectively `a op b`), for applying `op` on the contents of cells `a` and `b`, interpreted as 64-bit unsigned integers, with alignment `b8`.
- `a ++ b` is the field aligned binary concatenation of `a` and `b`.
- `a == b` for checking two cells for equality, at least one of which must contain at most 64 bytes of data
- `a & b`, `a | b`, `!a` are processed as boolean and, or, and not over the contents of cells `a` and maybe `b`. These must encode 1 or 0.
- `typeof(a)` returns a tag representing the type of a state value:
  - `<a: b>`: 0
  - `null`: 1
  - `Map { ... }`: 2
  - `Array(n) { ... }`: 3 + n * 32
  - `MerkleTree(n) { ... }`: 4 + n * 32
- `size(a)` returns the number of non-null entries is a `Map`, `n` for an `Array(n)` or `MerkleTree(n)`.
- `has_key(a, b)` returns `true` if `b` is a key to a non-null value in the `Map` `a`.
- `new ty` creates a new instance of a state value according to the tag `ty` (as returned by `typeof`):
  - cell: Containing the empty value.
  - `null` for itself
  - `Map`: The empty map
  - `Array(n)`: An array on `n` `null`s
  - `MerkleTree(n)`: A blank Merkle tree
- `a.get(b, cached)` retrieves the sub-item indexed with `b`. If the sub-item is *not* loaded in memory, *and* `cached` is `true`, this command fails. For different `a`:
  - `a: Map`, the value stored at the key `b`
  - `a: Array(n)`, the value at the index `b` < n

Feedback

- `rem(a, b, cached)` removes the sub-item indexed (as in `get`) with `b` from `a`. If the sub-item is *not* loaded in memory, *and* `cached` is `true`, this command fails.
- `ins(a, b, cached, c)` inserts `c` as a sub-item into `a` at index `c`. If the path for this index is *not* loaded in memory, *and* `cached` is `true`, this command fails.
- `root(a)` outputs the Merkle-tree root of the `MerkleTree(n)` `a`.

Context and effects

The `context` is an `Array(_)`, with the following entries, in order:

> CAUTION
>
> Currently, only the first two of these are correctly initialized!

1. A `Cell` containing the 256-bit aligned current contract's address.
2. A `Map` from `CoinCommitment` keys to 64-bit aligned Merkle tree indicies, for all newly allocated coins.
3. A `Cell` containing the block's 64-bit aligned seconds since the UNIX epoch approximation.
4. A `Cell` containing the block's 32-bit aligned seconds indicating the maximum amount that the former value may diverge.
5. A `Cell` containing the block's 256-bit hash.

This list may be extended in the future in a minor version increment.

The `effects` is an `Array(_)`, with the following entries, in order:

1. A `Map` from `Nullifier`s to `null`s, representing a set of claimed nullifiers.
2. A `Map` from `CoinCommitment`s to `null`s, representing a set of received coins claimed.
3. A `Map` from `CoinCommitment`s to `null`s, representing a set of spent coins claimed.

Feedback

4. A `Map` from `(Address, Bytes(32), Field)` to `null`, representing the contract calls claimed.
5. A `Map` from `Bytes(32)` to cells of `u64`, representing coins minted for any specialization hash.

This list may be extended in the future in a minor version increment.

`effects` is initialized to `[{}, {}, {}, {}, {}]`.

All of `context` and `effects` may be considered cached. To prevent cheaply copying data into the contract state with as little as two opcodes, both are flagged as *weak*, and any operations performed with them. If the final `state'` is tainted, the transaction fails, preventing this from being directly copied into the contract's state.

Feedback