9.4.2. *Exceptional Halting.* The exceptional halting function Z is defined as:

(149)
$$\begin{aligned} Z(\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I) \equiv\ & \boldsymbol{\mu}_g < C(\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I) \quad \vee \\ & \delta_w = \varnothing \quad \vee \\ & \|\boldsymbol{\mu}_\mathbf{s}\| < \delta_w \quad \vee \\ & (w = \text{JUMP} \ \wedge \ \boldsymbol{\mu}_\mathbf{s}[0] \notin D(I_\mathbf{b})) \quad \vee \\ & (w = \text{JUMPI} \ \wedge \ \boldsymbol{\mu}_\mathbf{s}[1] \neq 0 \ \wedge \\ & \quad \boldsymbol{\mu}_\mathbf{s}[0] \notin D(I_\mathbf{b})) \quad \vee \\ & (w = \text{RETURNDATACOPY} \ \wedge \\ & \quad \boldsymbol{\mu}_\mathbf{s}[1] + \boldsymbol{\mu}_\mathbf{s}[2] > \|\boldsymbol{\mu}_\mathbf{o}\|) \quad \vee \\ & \|\boldsymbol{\mu}_\mathbf{s}\| - \delta_w + \alpha_w > 1024 \quad \vee \\ & (\neg I_\mathrm{w} \ \wedge \ W(w, \boldsymbol{\mu})) \quad \vee \\ & (w = \text{SSTORE} \ \wedge \ \boldsymbol{\mu}_g \leqslant G_\mathrm{callstipend}) \end{aligned}$$

where

$$(150)\quad \begin{aligned} W(w, \boldsymbol{\mu}) \equiv\ & w \in \{\text{CREATE}, \text{CREATE2}, \text{SSTORE}, \\ & \text{SELFDESTRUCT}\} \ \vee \\ & \text{LOG0} \leq w \ \wedge \ w \leq \text{LOG4} \quad \vee \\ & w = \text{CALL} \ \wedge \ \boldsymbol{\mu}_\mathbf{s}[2] \neq 0 \end{aligned}$$

This states that the execution is in an exceptional halting state if there is insufficient gas, if the instruction is invalid (and therefore its δ subscript is undefined), if there are insufficient stack items, if a JUMP/JUMPI destination is invalid, the new stack size would be larger than 1024 or state modification is attempted during a static call. The astute reader will realise that this implies that no instruction can, through its execution, cause an exceptional halt. Also, the execution is in an exceptional halting state if the gas left prior to executing an SSTORE instruction is less than or equal to the call stipend $G_\mathrm{callstipend}$ – see EIP-2200 by Tang [2019] for more information.

9.4.3. *Jump Destination Validity.* We previously used D as the function to determine the set of valid jump destinations given the code that is being run. We define this as any position in the code occupied by a JUMPDEST instruction.

All such positions must be on valid instruction boundaries, rather than sitting in the data portion of PUSH operations and must appear within the explicitly defined portion of the code (rather than in the implicitly defined STOP operations that trail it).

Formally:

$$(151)\qquad D(\mathbf{c}) \equiv D_\mathrm{J}(\mathbf{c}, 0)$$

where:

(152)
$$D_\mathrm{J}(\mathbf{c}, i) \equiv \begin{cases} \{\} & \text{if} \quad i \geqslant \|\mathbf{c}\| \\ \{i\} \cup D_\mathrm{J}(\mathbf{c}, N(i, \mathbf{c}[i])) & \\ \qquad \text{if} \quad \mathbf{c}[i] = \text{JUMPDEST} & \\ D_\mathrm{J}(\mathbf{c}, N(i, \mathbf{c}[i])) & \text{otherwise} \end{cases}$$

where N is the next valid instruction position in the code, skipping the data of a PUSH instruction, if any:

(153)
$$N(i, w) \equiv \begin{cases} i + w - \text{PUSH1} + 2 & \\ \qquad \text{if} \quad w \in [\text{PUSH1}, \text{PUSH32}] & \\ i + 1 & \text{otherwise} \end{cases}$$

9.4.4. *Normal Halting.* The normal halting function H is defined:

(154)
$$H(\boldsymbol{\mu}, I) \equiv \begin{cases} H_\text{RETURN}(\boldsymbol{\mu}) & \text{if} \quad w \in \{\text{RETURN}, \text{REVERT}\} \\ () & \text{if} \quad w \in \{\text{STOP}, \text{SELFDESTRUCT}\} \\ \varnothing & \text{otherwise} \end{cases}$$

The data-returning halt operations, RETURN and REVERT, have a special function H_RETURN. Note also the difference between the empty sequence and the empty set as discussed here.

9.5. **The Execution Cycle.** Stack items are added or removed from the left-most, lower-indexed portion of the series; all other items remain unchanged:

$$\begin{aligned} &(155) & O\big((\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I)\big) &\equiv (\boldsymbol{\sigma}', \boldsymbol{\mu}', A', I) \\ &(156) & \Delta &\equiv \alpha_w - \delta_w \\ &(157) & \|\boldsymbol{\mu}'_\mathbf{s}\| &\equiv \|\boldsymbol{\mu}_\mathbf{s}\| + \Delta \\ &(158) & \forall x \in [\alpha_w, \|\boldsymbol{\mu}'_\mathbf{s}\|) : \boldsymbol{\mu}'_\mathbf{s}[x] &\equiv \boldsymbol{\mu}_\mathbf{s}[x - \Delta] \end{aligned}$$

The gas is reduced by the instruction's gas cost and for most instructions, the program counter increments on each cycle, for the three exceptions, we assume a function J, subscripted by one of two instructions, which evaluates to the according value:

$$(159)\qquad \boldsymbol{\mu}'_\mathrm{g} \equiv \boldsymbol{\mu}_\mathrm{g} - C(\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I)$$

$$(160)\qquad \boldsymbol{\mu}'_\mathrm{pc} \equiv \begin{cases} J_\text{JUMP}(\boldsymbol{\mu}) & \text{if} \quad w = \text{JUMP} \\ J_\text{JUMPI}(\boldsymbol{\mu}) & \text{if} \quad w = \text{JUMPI} \\ N(\boldsymbol{\mu}_\mathrm{pc}, w) & \text{otherwise} \end{cases}$$

In general, we assume the memory, accrued substate and system state do not change:

$$\begin{aligned} &(161) & \boldsymbol{\mu}'_\mathbf{m} &\equiv \boldsymbol{\mu}_\mathbf{m} \\ &(162) & \boldsymbol{\mu}'_\mathrm{i} &\equiv \boldsymbol{\mu}_\mathrm{i} \\ &(163) & A' &\equiv A \\ &(164) & \boldsymbol{\sigma}' &\equiv \boldsymbol{\sigma} \end{aligned}$$

However, instructions do typically alter one or several components of these values. Altered components listed by instruction are noted in Appendix H, alongside values for α and δ and a formal description of the gas requirements.

10. Blocktree to Blockchain

The canonical blockchain is a path from root to leaf through the entire block tree. In order to have consensus over which path it is, conceptually we identify the path that has had the most computation done upon it, or, the *heaviest* path. Clearly one factor that helps determine the heaviest path is the block number of the leaf, equivalent to the number of blocks, not counting the unmined genesis block, in the path. The longer the path, the greater the total mining effort that must have been done in order to arrive at the leaf. This is akin to existing schemes, such as that employed in Bitcoin-derived protocols.

Since a block header includes the difficulty, the header alone is enough to validate the computation done. Any block contributes toward the total computation or *total difficulty* of a chain.