The EVM Jello Paper Documentation

Release 0.1

The KEVM Team

Jul 29, 2018

CONTENTS

The Jello Paper is an attempt at defining the EVM semantics using the KEVM project. Unlike the Yellow Paper, the Jello Paper is an executable semantics, and can provide a full EVM interpreter usable for testing contracts, analyzing gas usage, verifying contracts correct, and a wide range of other tasks as specified in the technical report on KEVM.

The KEVM semantics described by the Jello Paper is the first machine-executable, mathematically formal, human readable, and complete semantics of the EVM. KEVM is capable of passing the full EVM VMTests and General-StateTests testing suites, and can also be used in smart contract formal verification, debugging, and more. The Jello Paper (this document) is automatically generated from the K definition of the KEVM semantics.

Start by choosing any section below or in the sidebar.

CHAPTER

ONE

KEVM: SEMANTICS OF EVM IN K

In this repository we provide a model of the EVM in K.

1.1 Documentation/Support

These may be useful for learning KEVM and K (newest to oldest):

- Jello Paper, generated using Sphinx Documentation Generation.
- 20 minute tour of the semantics at 2017 Devcon3.
- KEVM 1.0 technical report, especially sections 3 and 5.

To get support for KEVM, please join our Riot Room.

1.2 Installing/Building

1.2.1 K Backends

There are two backends of K available, the OCAML backend for concrete execution and the Java backend for symbolic reasoning and proofs. This repository generates the build-products for both backends in `.build/java/` and `.build/ocaml/`.

1.2.2 System Dependencies

The following are needed for building/running KEVM:

- Pandoc >= 1.17 is used to generate the `*.k` files from the `*.md` files.
- GNU Bison, Flex, and Autoconf.
- GNU libmpfr and libtool.
- Java 8 JDK (eg. OpenJDK)
- Opam, **important**: Ubuntu users prior to 15.04 **must** build from source, as the Ubuntu install for 14.10 and prior is broken. `opam repository` also requires `rsync`.

On Ubuntu >= 15.04 (for example):

```
sudo apt-get install make gcc maven openjdk-8-jdk flex opam pkg-config libmpfr-dev␣
↪autoconf libtool pandoc zlib1g-dev
```

To run proofs, you will also need Z3 prover; on Ubuntu:

```
sudo apt-get install z3 libz3-dev
```

On ArchLinux:

```
sudo pacman -S  base-devel rsync opam pandoc jre8-openjdk mpfr maven z3
```

On OSX, using Homebrew, after installing the command line tools package:

```
brew tap caskroom/cask caskroom/version
brew cask install java8
brew install automake libtool gmp mpfr pkg-config pandoc maven opam z3
```

NOTE: a previous version of these instructions required the user to run `brew link flex --force`. After fetching this revision, you should first run `brew unlink flex`, as it is no longer necessary and will cause an error if you have the homebrew version of flex installed instead of the xcode command line tools version.

1.2.3 Building

After installing the above dependencies, the following command will build submodule dependencies and then KEVM:

```
make deps
make
```

1.3 This Repository

The following files constitute the KEVM semantics:

- krypto.md sets up some basic cryptographic primitives.
- data.md provides the (functional) data of EVM (256 bit words, wordstacks, etc...).
- evm.md is the main KEVM semantics, containing the configuration and transition rules of EVM.

These additional files extend the semantics to make the repository more useful:

- driver.md is an execution harness for KEVM, providing a simple language for describing tests/programs.
- analysis.md contains any automated analysis tools we develop.
- edsl.md defines high-level notations of eDSL, a domain-specific language for EVM specifications, for formal verification of EVM bytecode using K Reachability Logic Prover.

1.4 Example Usage

After building the definition, you can run the definition using `./kevm`. Read the `./kevm` script for examples of the actual invocations of `krun` that `./kevm` makes.

Run the file `tests/ethereum-tests/VMTests/vmArithmeticTest/add0.json`:

```
./kevm run tests/ethereum-tests/VMTests/vmArithmeticTest/add0.json
```

Run the same file as a test:

```
./kevm test tests/ethereum-tests/VMTests/vmArithmeticTest/add0.json
```

To run proofs, you can similarly use `./kevm`. For example, to prove the specification `tests/proofs/specs/vyper-erc20/totalSupply-spec.k`:

```
./kevm prove tests/proofs/specs/vyper-erc20/totalSupply-spec.k
```

Finally, if you want to debug a given program (by stepping through its execution), you can use the `debug` option:

```
./kevm debug tests/ethereum-tests/VMTests/vmArithmeticTest/add0.json
...
KDebug> s
1 Step(s) Taken.
KDebug> p
... Big Configuration Here ...
KDebug>
```

1.5 Running Tests

The tests are run using the supplied `Makefile`. First, run `make split-tests` to generate some of the tests from the markdown files.

The following subsume all other tests:

- `make test`: All of the quick tests.
- `make test-all`: All of the quick and slow tests.

These are the individual test-suites (all of these can be suffixed with `-all` to also run slow tests):

- `make test-vm`: VMTests from the Ethereum Test Set.
- `make test-bchain`: Subset of BlockchainTests from the Ethereum Test Set.
- `make test-proof`: Proofs from the Verified Smart Contracts.
- `make test-interactive`: Tests of the `./kevm` command and of analysis tools.

1.6 Media

This repository can build two pieces of documentation for you, the Jello Paper and the 2017 Devcon3 presentation.

1.6.1 System Dependencies

If you also want to build the Jello Paper, you'll additionally need:

- Sphinx Documentation Generation tool, and
- The K Editor Support Python `pygments` package.

```
sudo apt-get install python-pygments python-sphinx python-recommonmark
git clone 'https://github.com/kframework/k-editor-support'
cd k-editor-support/pygments
easy_install --user .
```

For the 2017 Devcon3 presentation, you'll need `pdflatex`, commonly provided with `texlive-full`.

```
sudo apt-get install texlive-full
```

1.6.2 Building

The Makefile supplies targets for building:

- All media in this list: `make media`
- Jello Paper documentation: `make sphinx`
- 2017 Devcon3 presentation: `make 2017-devcon3`

1.7 Contributing

Any pull requests into this repository will not be reviewed until at least some conditions are met. Here we'll accumulate the standards that this repository is held to.

Code style guidelines, while somewhat subjective, will still be inspected before going to review. In general, read the rest of the definition for examples about how to style new K code; we collect a few common flubs here.

Writing tests and more contract proofs is **always** appreciated. Tests can come in the form of proofs done over contracts too :).

1.7.1 Hard - Every Commit

These are hard requirements (**must** be met before review), and they **must** be true for **every** commit in the PR.

- If a new feature is introduced in the PR, and later a bug is fixed in the new feature, the bug fix must be squashed back into the feature introduction. The *only* exceptions to this are if you want to document the bug because it was quite tricky or is something you believe should be fixed about K. In these exceptional cases, place the bug-fix commit directly after the feature introduction commit and leave useful commit messages. In addition, mark the feature introduction commit with `[skip-ci]` if tests will fail on that commit so that we know not to waste time testing it.
- No tab characters, 4 spaces instead. Linux-style line endings; if you're on a Windows machine make sure to run `dos2unix` on the files. No whitespace at the end of any lines.

1.7.2 Hard - PR Tip

These are hard requirements (**must** be met before review), but they only have to be true for the tip of the PR before review.

- Every test in the repository must pass. We will test this with `make split-tests ; make test -j12`.

1.7.3 Soft - Every Commit

These are soft requirements (review **may** start without these being met), and they will be considered for **every** commit in the PR.

- Comments do not live in the K code blocks, but rather in the surrounding Markdown (unless there is a really good reason to localize the comment).

- You should consider prefixing "internal" symbols (symbols that a user would not write in a program) with a hash (#).
- Place a line of - after each block of syntax declarations.

```
    syntax Foo ::= "newSymbol"
 // --------------------------
    rule <k> newSymbol => . ... </k>
```

  Notice that if there are rules immediately following the syntax declaration, a commented-out line of - is inserted afterward. Notice that the width of the line of - matches that of the preceding line.

- Place spaces around parentheses and commas in K's pretty functional-style syntax declarations.

```
    syntax Foo ::= newFunctionalSyntax ( Int , String )
 // ---------------------------------------------------
```

- When multiple structurally-similar rules are present, line up as much as possible (and makes sense).

```
    rule <k> #do1       => . ... </k> <cell1> not-done => done        </cell1>
    rule <k> #do1Longer => . ... </k> <cell1> not-done => done-longer </cell1>

    rule <k> #do2     => . ... </k> <cell2> not-done => done2 </cell2>
    rule <k> #doShort => . ... </k> <cell2> nd       => done2 </cell2>
```

  This makes it simpler to make changes to entire groups of rules at a time using sufficiently modern editors. Notice that if we break alignment (eg. from the `#do1` group above to the `#do2` group), we put an extra line between the groups of rules.

- Line up the `r` in `requires` with the `l` in `rule` (if it's not all on one line). Similarly, line up the end of `andBool` for extra side-conditions with the end of `requires`.

```
    rule <k> A => B ... </k>
         SOME_LARGE_CONFIGURATION

      requires A > 3
       andBool isPrime(A)
```

CHAPTER TWO

RESOURCES

- EVM Yellowpaper: Original specification of EVM.
- LEM Semantics of EVM

For more information about The K Framework, refer to these sources:

- The K Tutorial
- Semantics-Based Program Verifiers for All Languages
- Reachability Logic Resources
- Matching Logic Resources
- Logical Frameworks: Discussion of logical frameworks.

CHAPTER

THREE

EVM EXECUTION

3.1 Overview

The EVM is a stack machine over some simple opcodes. Most of the opcodes are "local" to the execution state of the machine, but some of them must interact with the world state. This file only defines the local execution operations, the file `driver.md` will define the interactions with the world state.

```
requires "data.k"
requires "network.k"

module EVM
    imports STRING
    imports EVM-DATA
    imports NETWORK
```

3.2 Configuration

The configuration has cells for the current account id, the current opcode, the program counter, the current gas, the gas price, the current program, the word stack, and the local memory. In addition, there are cells for the callstack and execution substate.

We've broken up the configuration into two components; those parts of the state that mutate during execution of a single transaction and those that are static throughout. In the comments next to each cell, we've marked which component of the YellowPaper state corresponds to each cell.

```
    configuration
      <k> $PGM:EthereumSimulation </k>
      <exit-code exit=""> 1 </exit-code>
      <mode> $MODE:Mode </mode>
      <schedule> $SCHEDULE:Schedule </schedule>
      <analysis> .Map </analysis>

      <ethereum>

        // EVM Specific
        // ============

        <evm>

          // Mutable during a single transaction
          // -----------------------------------
```

(continues on next page)

(continued from previous page)

```
        <output>          .WordStack  </output>          // H_RETURN
        <statusCode>      .StatusCode </statusCode>
        <callStack>       .List       </callStack>
        <interimStates>   .List       </interimStates>
        <touchedAccounts> .Set        </touchedAccounts>

        <callState>
          <program>      .Map       </program>           // I_b
          <programBytes> .WordStack </programBytes>

          // I_*
          <id>        0          </id>                   // I_a
          <caller>    0          </caller>               // I_s
          <callData>  .WordStack </callData>             // I_d
          <callValue> 0          </callValue>            // I_v

          // \mu_*
          <wordStack>   .WordStack </wordStack>          // \mu_s
          <localMem>    .Map       </localMem>           // \mu_m
          <pc>          0          </pc>                 // \mu_pc
          <gas>         0          </gas>                // \mu_g
          <memoryUsed>  0          </memoryUsed>         // \mu_i
          <previousGas> 0          </previousGas>

          <static>    false </static>
          <callDepth> 0     </callDepth>
        </callState>

        // A_* (execution substate)
        <substate>
          <selfDestruct> .Set  </selfDestruct>           // A_s
          <log>          .List </log>                    // A_l
          <refund>       0     </refund>                 // A_r
        </substate>

        // Immutable during a single transaction
        // -------------------------------------

        <gasPrice> 0 </gasPrice>                         // I_p
        <origin>   0 </origin>                           // I_o

        // I_H* (block information)
        <previousHash>     0          </previousHash>     // I_Hp
        <ommersHash>       0          </ommersHash>       // I_Ho
        <coinbase>         0          </coinbase>         // I_Hc
        <stateRoot>        0          </stateRoot>        // I_Hr
        <transactionsRoot> 0          </transactionsRoot> // I_Ht
        <receiptsRoot>     0          </receiptsRoot>     // I_He
        <logsBloom>        .WordStack </logsBloom>        // I_Hb
        <difficulty>       0          </difficulty>       // I_Hd
        <number>           0          </number>           // I_Hi
        <gasLimit>         0          </gasLimit>         // I_Hl
        <gasUsed>          0          </gasUsed>          // I_Hg
        <timestamp>        0          </timestamp>        // I_Hs
        <extraData>        .WordStack </extraData>        // I_Hx
        <mixHash>          0          </mixHash>          // I_Hm
```

(continues on next page)

(continued from previous page)

```
          <blockNonce>       0          </blockNonce>       // I_Hn

          <ommerBlockHeaders> [ .JSONList ] </ommerBlockHeaders>
          <blockhash>          .List         </blockhash>

        </evm>

        // Ethereum Network
        // ================

        <network>

          // Accounts Record
          // ---------------

          <activeAccounts> .Set </activeAccounts>
          <accounts>
            <account multiplicity="*" type="Map">
              <acctID>  0                      </acctID>
              <balance> 0                      </balance>
              <code>    .WordStack:AccountCode </code>
              <storage> .Map                   </storage>
              <nonce>   0                      </nonce>
            </account>
          </accounts>

          // Transactions Record
          // -------------------

          <txOrder>   .List </txOrder>
          <txPending> .List </txPending>

          <messages>
            <message multiplicity="*" type="Map">
              <msgID>      0          </msgID>
              <txNonce>    0          </txNonce>            // T_n
              <txGasPrice> 0          </txGasPrice>         // T_p
              <txGasLimit> 0          </txGasLimit>         // T_g
              <to>         .Account   </to>                 // T_t
              <value>      0          </value>              // T_v
              <sigV>       0          </sigV>               // T_w
              <sigR>       .WordStack </sigR>               // T_r
              <sigS>       .WordStack </sigS>               // T_s
              <data>       .WordStack </data>               // T_i/T_e
            </message>
          </messages>

        </network>

      </ethereum>

    syntax EthereumSimulation
    syntax AccountCode ::= WordStack
 // --------------------------------
```

3.3 Modal Semantics

Our semantics is modal, with the initial mode being set on the command line via -cMODE=EXECMODE.

- NORMAL executes as a client on the network would.
- VMTESTS skips CALL* and CREATE operations.

```
    syntax Mode ::= "NORMAL"   [klabel(NORMAL)]
                  | "VMTESTS" [klabel(VMTESTS)]
```

- #setMode_ sets the mode to the supplied one.

```
    syntax InternalOp ::= "#setMode" Mode
 // ------------------------------------
    rule <k> #setMode EXECMODE => . ... </k> <mode> _ => EXECMODE </mode>
```

3.4 State Stacks

3.4.1 The CallStack

The callStack cell stores a list of previous VM execution states.

- #pushCallStack saves a copy of VM execution state on the callStack.
- #popCallStack restores the top element of the callStack.
- #dropCallStack removes the top element of the callStack.

```
    syntax InternalOp ::= "#pushCallStack"
 // --------------------------------------
    rule <k> #pushCallStack => . ... </k>
         <callStack> (.List => ListItem(CALLSTATE)) ... </callStack>
         <callState> CALLSTATE </callState>

    syntax InternalOp ::= "#popCallStack"
 // -------------------------------------
    rule <k> #popCallStack => . ... </k>
         <callStack>   (ListItem(CALLSTATE) => .List) ... </callStack>
         <callState> _ => CALLSTATE </callState>

    syntax InternalOp ::= "#dropCallStack"
 // --------------------------------------
    rule <k> #dropCallStack => . ... </k>
         <callStack> (ListItem(_) => .List) ... </callStack>
```

3.4.2 The StateStack

The interimStates cell stores a list of previous world states.

- #pushWorldState stores a copy of the current accounts and the substate at the top of the interimStates cell.
- #popWorldState restores the top element of the interimStates.
- #dropWorldState removes the top element of the interimStates.

```
    syntax Accounts ::= "{" AccountsCellFragment "|" Set "|" SubstateCellFragment "}"
 // ------------------------------------------------------------------------------------

    syntax InternalOp ::= "#pushWorldState"
 // ---------------------------------------
    rule <k> #pushWorldState => .K ... </k>
         <interimStates> (.List => ListItem({ ACCTDATA | ACCTS | SUBSTATE })) ... </
↪interimStates>
         <activeAccounts> ACCTS    </activeAccounts>
         <accounts>       ACCTDATA </accounts>
         <substate>       SUBSTATE </substate>

    syntax InternalOp ::= "#popWorldState"
 // --------------------------------------
    rule <k> #popWorldState => .K ... </k>
         <interimStates> (ListItem({ ACCTDATA | ACCTS | SUBSTATE }) => .List) ... </
↪interimStates>
         <activeAccounts> _ => ACCTS    </activeAccounts>
         <accounts>       _ => ACCTDATA </accounts>
         <substate>       _ => SUBSTATE </substate>

    syntax InternalOp ::= "#dropWorldState"
 // ---------------------------------------
    rule <k> #dropWorldState => . ... </k> <interimStates> (ListItem(_) => .List) ...
↪</interimStates>
```

3.5 Control Flow

3.5.1 Exception Based

- `#halt` indicates end of execution. It will consume anything related to the current computation behind it on the `<k>` cell.
- `#end_` sets the `statusCode` then halts execution.

```
    syntax KItem ::= "#halt" | "#end" StatusCode
 // ---------------------------------------------
    rule <k> #end SC => #halt ... </k> <statusCode> _ => SC </statusCode>

    rule <k> #halt ~> (_:Int    => .) ... </k>
    rule <k> #halt ~> (_:OpCode => .) ... </k>
```

- `#?_:_?#` provides an "if-then-else" (choice):
  - If there is no exception, take the first branch.
  - Else, catch exception and take the second branch.

```
    syntax KItem ::= "#?" K ":" K "?#"
 // ----------------------------------
    rule <k> #? B1 : _  ?# => B1 ... </k>
    rule <statusCode> SC </statusCode>
         <k> #halt ~> #? B1 : B2 ?# => #if isExceptionalStatusCode(SC) #then B2 #else␣
↪B1 #fi ~> #halt ... </k>
```

3.6 OpCode Execution

3.6.1 Execution Macros

- `#execute` calls `#next` repeatedly until it recieves an `#end`.
- `#execTo` executes until the next opcode is one of the specified ones.

```
    syntax KItem ::= "#execute"
 // ----------------------------
    rule <k> (. => #next) ~> #execute ... </k> [tag(step)]
    rule <k> #halt ~> (#execute => .) ... </k> [tag(step)]

    syntax InternalOp ::= "#execTo" Set
 // ------------------------------------
    rule <k> (. => #next) ~> #execTo OPS ... </k>
         <pc> PCOUNT </pc>
         <program> ... PCOUNT |-> OP ... </program>
      requires notBool (OP in OPS)

    rule <k> #execTo OPS => . ... </k>
         <pc> PCOUNT </pc>
         <program> ... PCOUNT |-> OP ... </program>
      requires OP in OPS

    rule <k> #execTo OPS => #end EVMC_SUCCESS ... </k>
         <pc> PCOUNT </pc>
         <program> PGM </program>
      requires notBool PCOUNT in keys(PGM)
```

Execution follows a simple cycle where first the state is checked for exceptions, then if no exceptions will be thrown the opcode is run. When the `#next` operator cannot lookup the next opcode, it assumes that the end of execution has been reached.

```
    syntax InternalOp ::= "#next"
 // -----------------------------
    rule <k> #next => #end EVMC_SUCCESS ... </k>
         <pc> PCOUNT </pc>
         <program> PGM </program>
         <output> _ => .WordStack </output>
      requires notBool (PCOUNT in_keys(PGM))
```

3.6.2 Single Step

The `#next` operator executes a single step by:

1. performing some quick checks for exceptional opcodes,
2. executes the opcode if it is not immediately exceptional,
3. increments the program counter, and finally
4. reverts state if any of the above steps threw an exception.

```
    rule <mode> EXECMODE </mode>
         <k> #next
```

(continues on next page)

(continued from previous page)

```
          => #exceptional? [ OP ]
          ~> #load         [ OP ]
          ~> #exec         [ OP ]
          ~> #pc           [ OP ]
         ...
         </k>
         <pc> PCOUNT </pc>
         <program> ... PCOUNT |-> OP ... </program>
      requires EXECMODE in (SetItem(NORMAL) SetItem(VMTESTS))
```

3.6.3 Exceptional OpCodes

- `#exceptional?` checks if the operator is invalid and will not cause `wordStack` size issues (this implements the function `Z` in the YellowPaper, section 9.4.2).

```
    syntax InternalOp ::= "#exceptional?" "[" OpCode "]"
 // ----------------------------------------------------
    rule <k> #exceptional? [ OP ]
          => #invalid?     [ OP ]
          ~> #stackNeeded? [ OP ]
          ~> #static?      [ OP ]
         ...
         </k>
```

- `#invalid?` checks if it's the designated invalid opcode or some undefined opcode.

```
    syntax InternalOp ::= "#invalid?" "[" OpCode "]"
 // ------------------------------------------------
    rule <k> #invalid? [ INVALID      ] => #end EVMC_INVALID_INSTRUCTION   ... </k>
    rule <k> #invalid? [ UNDEFINED(_) ] => #end EVMC_UNDEFINED_INSTRUCTION ... </k>
    rule <k> #invalid? [ OP           ] => .                               ... </k>
↪requires notBool isInvalidOp(OP)
```

- `#stackNeeded?` checks that the stack will be not be under/overflown.
- `#stackNeeded`, `#stackAdded`, and `#stackDelta` are helpers for deciding `#stackNeeded?`.

```
    syntax InternalOp ::= "#stackNeeded?" "[" OpCode "]"
 // ----------------------------------------------------
    rule <k> #stackNeeded? [ OP ] => #end EVMC_STACK_UNDERFLOW ... </k>
         <wordStack> WS </wordStack>
      requires #stackUnderflow(WS, OP)

    rule <k> #stackNeeded? [ OP ] => #end EVMC_STACK_OVERFLOW ... </k>
         <wordStack> WS </wordStack>
      requires #stackOverflow(WS, OP)

    rule <k> #stackNeeded? [ OP ] => . ... </k>
         <wordStack> WS </wordStack>
      requires notBool ( #stackUnderflow(WS, OP) orBool #stackOverflow(WS, OP) )

    syntax Bool ::= #stackUnderflow ( WordStack , OpCode ) [function]
                  | #stackOverflow  ( WordStack , OpCode ) [function]
 // -----------------------------------------------------------------
    rule #stackUnderflow(WS, OP) => #sizeWordStack(WS)                 <Int
↪#stackNeeded(OP)
```

(continues on next page)

(continued from previous page)

```
    rule #stackOverflow (WS, OP) => #sizeWordStack(WS) +Int #stackDelta(OP) >Int 1024

    syntax Int ::= #stackNeeded ( OpCode ) [function]
 // -------------------------------------------------
    rule #stackNeeded(PUSH(_, _))      => 0
    rule #stackNeeded(NOP:NullStackOp) => 0
    rule #stackNeeded(UOP:UnStackOp)   => 1
    rule #stackNeeded(BOP:BinStackOp)  => 2 requires notBool isLogOp(BOP)
    rule #stackNeeded(TOP:TernStackOp) => 3
    rule #stackNeeded(QOP:QuadStackOp) => 4
    rule #stackNeeded(DUP(N))          => N
    rule #stackNeeded(SWAP(N))         => N +Int 1
    rule #stackNeeded(LOG(N))          => N +Int 2
    rule #stackNeeded(CSOP:CallSixOp)  => 6
    rule #stackNeeded(COP:CallOp)      => 7 requires notBool isCallSixOp(COP)

    syntax Int ::= #stackAdded ( OpCode ) [function]
 // ------------------------------------------------
    rule #stackAdded(CALLDATACOPY)   => 0
    rule #stackAdded(RETURNDATACOPY) => 0
    rule #stackAdded(CODECOPY)       => 0
    rule #stackAdded(EXTCODECOPY)    => 0
    rule #stackAdded(POP)            => 0
    rule #stackAdded(MSTORE)         => 0
    rule #stackAdded(MSTORE8)        => 0
    rule #stackAdded(SSTORE)         => 0
    rule #stackAdded(JUMP)           => 0
    rule #stackAdded(JUMPI)          => 0
    rule #stackAdded(JUMPDEST)       => 0
    rule #stackAdded(STOP)           => 0
    rule #stackAdded(RETURN)         => 0
    rule #stackAdded(REVERT)         => 0
    rule #stackAdded(SELFDESTRUCT)   => 0
    rule #stackAdded(PUSH(_,_))      => 1
    rule #stackAdded(LOG(_))         => 0
    rule #stackAdded(SWAP(N))        => N
    rule #stackAdded(DUP(N))         => N +Int 1
    rule #stackAdded(OP)             => 1 [owise]

    syntax Int ::= #stackDelta ( OpCode ) [function]
 // ------------------------------------------------
    rule #stackDelta(OP) => #stackAdded(OP) -Int #stackNeeded(OP)
```

- `#static?` determines if the opcode should throw an exception due to the static flag.

```
    syntax InternalOp ::= "#static?" "[" OpCode "]"
 // -----------------------------------------------
    rule <k> #static? [ OP ] => .                          ... </k>
 ↪                <static> false </static>
    rule <k> #static? [ OP ] => .                          ... </k> <wordStack>
 ↪WS </wordStack> <static> true  </static> requires notBool #changesState(OP, WS)
    rule <k> #static? [ OP ] => #end EVMC_STATIC_MODE_VIOLATION ... </k> <wordStack>
 ↪WS </wordStack> <static> true  </static> requires         #changesState(OP, WS)
```

TODO: Investigate why using `[owise]` here for the `false` cases breaks the proofs. Alternatively, figure out how to make this go through with a boolean expression.

```
    syntax Bool ::= #changesState ( OpCode , WordStack ) [function]
 // --------------------------------------------------------------
    rule #changesState(LOG(_), _)               => true
    rule #changesState(SSTORE, _)               => true
    rule #changesState(CALL, _ : _ : VALUE : _) => VALUE =/=Int 0
    rule #changesState(CREATE, _)               => true
    rule #changesState(SELFDESTRUCT, _)         => true

    rule #changesState(DUP(_), _)         => false
    rule #changesState(SWAP(_), _)        => false
    rule #changesState(PUSH(_, _), _)     => false
    rule #changesState(STOP, _)           => false
    rule #changesState(ADD, _)            => false
    rule #changesState(MUL, _)            => false
    rule #changesState(SUB, _)            => false
    rule #changesState(DIV, _)            => false
    rule #changesState(SDIV, _)           => false
    rule #changesState(MOD, _)            => false
    rule #changesState(SMOD, _)           => false
    rule #changesState(ADDMOD, _)         => false
    rule #changesState(MULMOD, _)         => false
    rule #changesState(EXP, _)            => false
    rule #changesState(SIGNEXTEND, _)     => false
    rule #changesState(LT, _)             => false
    rule #changesState(GT, _)             => false
    rule #changesState(SLT, _)            => false
    rule #changesState(SGT, _)            => false
    rule #changesState(EQ, _)             => false
    rule #changesState(ISZERO, _)         => false
    rule #changesState(AND, _)            => false
    rule #changesState(EVMOR, _)          => false
    rule #changesState(XOR, _)            => false
    rule #changesState(NOT, _)            => false
    rule #changesState(BYTE, _)           => false
    rule #changesState(SHA3, _)           => false
    rule #changesState(ADDRESS, _)        => false
    rule #changesState(BALANCE, _)        => false
    rule #changesState(ORIGIN, _)         => false
    rule #changesState(CALLER, _)         => false
    rule #changesState(CALLVALUE, _)      => false
    rule #changesState(CALLDATALOAD, _)   => false
    rule #changesState(CALLDATASIZE, _)   => false
    rule #changesState(CALLDATACOPY, _)   => false
    rule #changesState(CODESIZE, _)       => false
    rule #changesState(CODECOPY, _)       => false
    rule #changesState(GASPRICE, _)       => false
    rule #changesState(EXTCODESIZE, _)    => false
    rule #changesState(EXTCODECOPY, _)    => false
    rule #changesState(RETURNDATASIZE, _) => false
    rule #changesState(RETURNDATACOPY, _) => false
    rule #changesState(BLOCKHASH, _)      => false
    rule #changesState(COINBASE, _)       => false
    rule #changesState(TIMESTAMP, _)      => false
    rule #changesState(NUMBER, _)         => false
    rule #changesState(DIFFICULTY, _)     => false
    rule #changesState(GASLIMIT, _)       => false
    rule #changesState(POP, _)            => false
```

(continues on next page)

(continued from previous page)

```
    rule #changesState(MLOAD, _)           => false
    rule #changesState(MSTORE, _)          => false
    rule #changesState(MSTORE8, _)         => false
    rule #changesState(SLOAD, _)           => false
    rule #changesState(JUMP, _)            => false
    rule #changesState(JUMPI, _)           => false
    rule #changesState(PC, _)              => false
    rule #changesState(MSIZE, _)           => false
    rule #changesState(GAS, _)             => false
    rule #changesState(JUMPDEST, _)        => false
    rule #changesState(CALLCODE, _)        => false
    rule #changesState(RETURN, _)          => false
    rule #changesState(DELEGATECALL, _)    => false
    rule #changesState(STATICCALL, _)      => false
    rule #changesState(REVERT, _)          => false
    rule #changesState(INVALID, _)         => false
    rule #changesState(UNDEFINED(_), _)    => false

    rule #changesState(ECREC, _)      => false
    rule #changesState(SHA256, _)     => false
    rule #changesState(RIP160, _)     => false
    rule #changesState(ID, _)         => false
    rule #changesState(MODEXP, _)     => false
    rule #changesState(ECADD, _)      => false
    rule #changesState(ECMUL, _)      => false
    rule #changesState(ECPAIRING, _) => false
```

3.6.4 Execution Step

- `#exec` will load the arguments of the opcode (it assumes `#stackNeeded?` is accurate and has been called) and trigger the subsequent operations.

```
    syntax InternalOp ::= "#exec" "[" OpCode "]"
 // --------------------------------------------
    rule <k> #exec [ OP ] => #gas [ OP ] ~> OP ... </k> requires isInternalOp(OP)␣
↪orBool isNullStackOp(OP) orBool isPushOp(OP)
```

Here we load the correct number of arguments from the `wordStack` based on the sort of the opcode. Some of them require an argument to be interpereted as an address (modulo 160 bits), so the `#addr?` function performs that check.

```
    syntax KItem  ::= OpCode
    syntax OpCode ::= NullStackOp | UnStackOp | BinStackOp | TernStackOp | QuadStackOp
                    | InvalidOp | StackOp | InternalOp | CallOp | CallSixOp | PushOp
 // --------------------------------------------------------------------------------

    syntax InternalOp ::= UnStackOp   Int
                        | BinStackOp  Int Int
                        | TernStackOp Int Int Int
                        | QuadStackOp Int Int Int Int
 // -------------------------------------------------
    rule <k> #exec [ UOP:UnStackOp   => UOP W0          ] ... </k> <wordStack> W0 :␣
↪WS          => WS </wordStack>
    rule <k> #exec [ BOP:BinStackOp  => BOP W0 W1       ] ... </k> <wordStack> W0 :␣
↪W1 : WS     => WS </wordStack>
    rule <k> #exec [ TOP:TernStackOp => TOP W0 W1 W2    ] ... </k> <wordStack> W0 :␣
↪W1 : W2 : WS     => WS </wordStack>
```

(continues on next page)

(continued from previous page)

```
    rule <k> #exec [ QOP:QuadStackOp => QOP W0 W1 W2 W3 ] ... </k> <wordStack> W0 :␣
↪W1 : W2 : W3 : WS => WS </wordStack>
```

StackOp is used for opcodes which require a large portion of the stack.

```
    syntax InternalOp ::= StackOp WordStack
 // --------------------------------------
    rule <k> #exec [ SO:StackOp => SO WS ] ... </k> <wordStack> WS </wordStack>
```

The CallOp opcodes all interperet their second argument as an address.

```
    syntax InternalOp ::= CallSixOp Int Int     Int Int Int Int
                        | CallOp    Int Int Int Int Int Int Int
 // -----------------------------------------------------------
    rule <k> #exec [ CSO:CallSixOp => CSO W0 W1    W2 W3 W4 W5 ] ... </k> <wordStack>␣
↪W0 : W1 : W2 : W3 : W4 : W5 : WS      => WS </wordStack>
    rule <k> #exec [ CO:CallOp     => CO  W0 W1 W2 W3 W4 W5 W6 ] ... </k> <wordStack>␣
↪W0 : W1 : W2 : W3 : W4 : W5 : W6 : WS => WS </wordStack>
```

3.6.5 Helpers

- #addr decides if the given argument should be interpreted as an address (given the opcode).
- #gas calculates how much gas this operation costs, and takes into account the memory consumed.

```
    syntax InternalOp ::= "#load" "[" OpCode "]"
 // --------------------------------------------
    rule <k> #load [ OP:OpCode ] => #loadAccount #addr(W0) ... </k>
         <wordStack> (W0 => #addr(W0)) : WS </wordStack>
      requires #addr?(OP)

    rule <k> #load [ OP:OpCode ] => #loadAccount #addr(W0) ~> #lookupCode #addr(W0) ..
↪. </k>
         <wordStack> (W0 => #addr(W0)) : WS </wordStack>
      requires #code?(OP)

    rule <k> #load [ OP:OpCode ] => #loadAccount #addr(W1) ~> #lookupCode #addr(W1) ..
↪. </k>
         <wordStack> W0 : (W1 => #addr(W1)) : WS </wordStack>
      requires isCallOp(OP) orBool isCallSixOp(OP)

    rule <k> #load [ CREATE ] => #loadAccount #newAddr(ACCT, NONCE) ... </k>
         <id> ACCT </id>
         <account>
           <acctID> ACCT </acctID>
           <nonce> NONCE </nonce>
           ...
         </account>

    rule <k> #load [ OP:OpCode ] => #lookupStorage ACCT W0 ... </k>
         <id> ACCT </id>
         <wordStack> W0 : WS </wordStack>
      requires OP ==K SSTORE orBool OP ==K SLOAD

    rule <k> #load [ OP:OpCode ] => . ... </k>
```

(continues on next page)

(continued from previous page)

```
      requires notBool (
        OP ==K CREATE   orBool
        OP ==K SLOAD    orBool
        OP ==K SSTORE   orBool
        isCallOp    (OP) orBool
        isCallSixOp(OP) orBool
        #addr?(OP)       orBool
        #code?(OP)
      )

    syntax Bool ::= "#addr?" "(" OpCode ")" [function]
 // --------------------------------------------------
    rule #addr?(BALANCE)      => true
    rule #addr?(SELFDESTRUCT) => true
    rule #addr?(OP)           => false requires (OP =/=K BALANCE) andBool (OP =/=K␣
↪SELFDESTRUCT)

    syntax Bool ::= "#code?" "(" OpCode ")" [function]
 // --------------------------------------------------
    rule #code?(EXTCODESIZE)  => true
    rule #code?(EXTCODECOPY)  => true
    rule #code?(OP)           => false requires (OP =/=K EXTCODESIZE) andBool (OP =/
↪=K EXTCODECOPY)

    syntax InternalOp ::= "#gas" "[" OpCode "]" | "#deductGas" | "#deductMemory"
 // ----------------------------------------------------------------------------
    rule <k> #gas [ OP ] => #memory(OP, MU) ~> #deductMemory ~> #gasExec(SCHED, OP)␣
↪~> #deductGas ... </k> <memoryUsed> MU </memoryUsed> <schedule> SCHED </schedule>

    rule <k> MU':Int ~> #deductMemory => #end EVMC_INVALID_MEMORY_ACCESS ... </k>␣
↪requires MU' >=Int pow256
    rule <k> MU':Int ~> #deductMemory => (Cmem(SCHED, MU') -Int Cmem(SCHED, MU)) ~>
↪#deductGas ... </k>
         <memoryUsed> MU => MU' </memoryUsed> <schedule> SCHED </schedule>
      requires MU' <Int pow256

    rule <k> G:Int ~> #deductGas => #end EVMC_OUT_OF_GAS ... </k> <gas> GAVAIL      ␣
↪         </gas> requires GAVAIL <Int G
    rule <k> G:Int ~> #deductGas => .                    ... </k> <gas> GAVAIL =>␣
↪GAVAIL -Int G </gas> <previousGas> _ => GAVAIL </previousGas> requires GAVAIL >=Int␣
↪G

    syntax Int ::= Cmem ( Schedule , Int ) [function, memo]
 // -------------------------------------------------------
    rule Cmem(SCHED, N) => (N *Int Gmemory < SCHED >) +Int ((N *Int N) /Int␣
↪Gquadcoeff < SCHED >)
```

3.6.6 Program Counter

All operators except for PUSH and JUMP* increment the program counter by 1. The arguments to PUSH must be skipped over (as they are inline), and the opcode JUMP already affects the program counter in the correct way.

- #pc calculates the next program counter of the given operator.

```
    syntax InternalOp ::= "#pc" "[" OpCode "]"
 // -------------------------------------------
    rule <k> #pc [ OP         ] => . ... </k> <pc> PCOUNT => PCOUNT +Int 1          </
↪pc> requires notBool (isPushOp(OP) orBool isJumpOp(OP))
    rule <k> #pc [ PUSH(N, _) ] => . ... </k> <pc> PCOUNT => PCOUNT +Int (1 +Int N) </
↪pc>
    rule <k> #pc [ OP         ] => . ... </k> requires isJumpOp(OP)

    syntax Bool ::= isJumpOp ( OpCode ) [function]
 // -----------------------------------------------
    rule isJumpOp(OP) => OP ==K JUMP orBool OP ==K JUMPI
```

3.6.7 Substate Log

During execution of a transaction some things are recorded in the substate log (Section 6.1 in YellowPaper). This is a right cons-list of `SubstateLogEntry` (which contains the account ID along with the specified portions of the `wordStack` and `localMem`).

```
    syntax SubstateLogEntry ::= "{" Int "|" List "|" WordStack "}" [klabel(logEntry)]
 // ---------------------------------------------------------------------------------
```

After executing a transaction, it's necessary to have the effect of the substate log recorded.

- `#finalizeTx` makes the substate log actually have an effect on the state.

```
    syntax InternalOp ::= #finalizeTx ( Bool )
 // ------------------------------------------
    rule <k> #finalizeTx(true) => . ... </k>
         <selfDestruct> .Set </selfDestruct>

    rule <k> (.K => #newAccount MINER) ~> #finalizeTx(_)... </k>
         <coinbase> MINER </coinbase>
         <activeAccounts> ACCTS </activeAccounts>
      requires notBool MINER in ACCTS

    rule <k> #finalizeTx(false) ... </k>
         <gas> GAVAIL => G*(GAVAIL, GLIMIT, REFUND) </gas>
         <refund> REFUND => 0 </refund>
         <txPending> ListItem(MSGID:Int) ... </txPending>
         <message>
            <msgID> MSGID </msgID>
            <txGasLimit> GLIMIT </txGasLimit>
            ...
         </message>
      requires REFUND =/=Int 0

    rule <k> #finalizeTx(false => true) ... </k>
         <origin> ORG </origin>
         <coinbase> MINER </coinbase>
         <gas> GAVAIL </gas>
         <refund> 0 </refund>
         <account>
           <acctID> ORG </acctID>
           <balance> ORGBAL => ORGBAL +Int GAVAIL *Int GPRICE </balance>
           ...
         </account>
```

(continues on next page)

(continued from previous page)

```
          <account>
            <acctID> MINER </acctID>
            <balance> MINBAL => MINBAL +Int (GLIMIT -Int GAVAIL) *Int GPRICE </balance>
            ...
          </account>
          <txPending> ListItem(TXID:Int) => .List ... </txPending>
          <message>
            <msgID> TXID </msgID>
            <txGasLimit> GLIMIT </txGasLimit>
            <txGasPrice> GPRICE </txGasPrice>
            ...
          </message>
       requires ORG =/=Int MINER

     rule <k> #finalizeTx(false => true) ... </k>
          <origin> ACCT </origin>
          <coinbase> ACCT </coinbase>
          <refund> 0 </refund>
          <account>
            <acctID> ACCT </acctID>
            <balance> BAL => BAL +Int GLIMIT *Int GPRICE </balance>
            ...
          </account>
          <txPending> ListItem(MsgId:Int) => .List ... </txPending>
          <message>
            <msgID> MsgId </msgID>
            <txGasLimit> GLIMIT </txGasLimit>
            <txGasPrice> GPRICE </txGasPrice>
            ...
          </message>

     rule <k> #finalizeTx(true) ... </k>
          <selfDestruct> ... (SetItem(ACCT) => .Set) </selfDestruct>
          <activeAccounts> ... (SetItem(ACCT) => .Set) </activeAccounts>
          <accounts>
            ( <account>
                <acctID> ACCT </acctID>
                ...
              </account>
           => .Bag
            )
            ...
          </accounts>
```

CHAPTER

FOUR

EVM PROGRAMS

4.1 Program Structure

Lists of opcodes form programs.

```
    syntax OpCodes ::= ".OpCodes" | OpCode ";" OpCodes
 // --------------------------------------------------
```

4.2 Converting to/from `Map` Representation

```
    syntax Map ::= #asMapOpCodes ( OpCodes )             [function]
                 | #asMapOpCodes ( Int , OpCodes , Map ) [function, klabel(
↪#asMapOpCodesAux)]
 // ----------------------------------------------------------------------------------
↪-------
    rule #asMapOpCodes( OPS::OpCodes ) => #asMapOpCodes(0, OPS, .Map)

    rule #asMapOpCodes( N , .OpCodes          , MAP ) => MAP
    rule #asMapOpCodes( N , OP:OpCode   ; OCS , MAP ) => #asMapOpCodes(N +Int 1, OCS,␣
↪MAP (N |-> OP)) requires notBool isPushOp(OP)
    rule #asMapOpCodes( N , PUSH(M, W) ; OCS , MAP ) => #asMapOpCodes(N +Int 1 +Int M,
↪ OCS, MAP (N |-> PUSH(M, W)))

    syntax OpCodes ::= #asOpCodes ( Map )                   [function]
                     | #asOpCodes ( Int , Map , OpCodes ) [function, klabel(
↪#asOpCodesAux)]
 // ----------------------------------------------------------------------------------
↪-----
    rule #asOpCodes(M) => #asOpCodes(0, M, .OpCodes)

    rule #asOpCodes(N, .Map,               OPS) => OPS
    rule #asOpCodes(N, N |-> OP         M, OPS) => #asOpCodes(N +Int 1,        M, OP ␣
↪       ; OPS) requires notBool isPushOp(OP)
    rule #asOpCodes(N, N |-> PUSH(S, W) M, OPS) => #asOpCodes(N +Int 1 +Int S, M,␣
↪PUSH(S, W) ; OPS)
```

4.3 EVM OpCodes

4.3.1 Internal Operations

These are just used by the other operators for shuffling local execution state around on the EVM.

- `#push` will push an element to the `wordStack` without any checks.
- `#setStack_` will set the current stack to the given one.

```
    syntax InternalOp ::= "#push" | "#setStack" WordStack
 // ------------------------------------------------------
    rule <k> W0:Int ~> #push => . ... </k> <wordStack> WS => W0 : WS </wordStack>
    rule <k> #setStack WS    => . ... </k> <wordStack> _  => WS      </wordStack>
```

- `#newAccount_` allows declaring a new empty account with the given address (and assumes the rounding to 160 bits has already occured). If the account already exists with non-zero nonce or non-empty code, an exception is thrown. Otherwise, if the account already exists, the storage is cleared.

```
    syntax InternalOp ::= "#newAccount" Int
 // ---------------------------------------
    rule <k> #newAccount ACCT => #end EVMC_ACCOUNT_ALREADY_EXISTS ... </k>
         <account>
           <acctID> ACCT  </acctID>
           <code>   CODE  </code>
           <nonce>  NONCE </nonce>
           ...
         </account>
      requires CODE =/=K .WordStack orBool NONCE =/=K 0

    rule <k> #newAccount ACCT => . ... </k>
         <account>
           <acctID>  ACCT       </acctID>
           <code>    .WordStack </code>
           <nonce>   0          </nonce>
           <storage> _ => .Map  </storage>
           ...
         </account>

    rule <k> #newAccount ACCT => . ... </k>
         <activeAccounts> ACCTS (.Set => SetItem(ACCT)) </activeAccounts>
         <accounts>
           ( .Bag
          => <account>
               <acctID>  ACCT       </acctID>
               ...
             </account>
           )
           ...
         </accounts>
      requires notBool ACCT in ACCTS
```

The following operations help with loading account information from an external running client. This minimizes the amount of information which must be stored in the configuration.

- `#loadAccount` queries for account data from the running client.
- `#lookupCode` loads the code of an account into the `<code>` cell.

- `#lookupStorage` loads the value of the specified storage key into the `<storage>` cell.

```
    syntax InternalOp ::= "#loadAccount"    Int
                        | "#lookupCode"     Int
                        | "#lookupStorage" Int Int
 // ----------------------------------------------
```

In `standalone` mode, the semantics assumes that all relevant account data is already loaded into memory.

```
    rule <k> #loadAccount   _   => . ... </k>
    rule <k> #lookupCode    _   => . ... </k>
    rule <k> #lookupStorage _ _ => . ... </k>
```

In `node` mode, the semantics are given in terms of an external call to a running client.

```
    rule <k> #lookupStorage ACCT INDEX => . ... </k>
         <account>
           <acctID> ACCT </acctID>
           <storage> ... INDEX |-> _ ... </storage>
           ...
         </account>
```

- `#transferFunds` moves money from one account into another, creating the destination account if it doesn't exist.

```
    syntax InternalOp ::= "#transferFunds" Int Int Int
 // --------------------------------------------------
    rule <k> #transferFunds ACCT ACCT VALUE => . ... </k>
         <account>
           <acctID> ACCT </acctID>
           <balance> ORIGFROM </balance>
           ...
         </account>
      requires VALUE <=Int ORIGFROM

    rule <k> #transferFunds ACCTFROM ACCTTO VALUE => . ... </k>
         <account>
           <acctID> ACCTFROM </acctID>
           <balance> ORIGFROM => ORIGFROM -Word VALUE </balance>
           ...
         </account>
         <account>
           <acctID> ACCTTO </acctID>
           <balance> ORIGTO => ORIGTO +Word VALUE </balance>
           ...
         </account>
      requires ACCTFROM =/=K ACCTTO andBool VALUE <=Int ORIGFROM

    rule <k> #transferFunds ACCTFROM ACCTTO VALUE => #end EVMC_BALANCE_UNDERFLOW ...
↪</k>
         <account>
           <acctID> ACCTFROM </acctID>
           <balance> ORIGFROM </balance>
           ...
         </account>
      requires VALUE >Int ORIGFROM

    rule <k> (. => #newAccount ACCTTO) ~> #transferFunds ACCTFROM ACCTTO VALUE ... </
↪k>
```

(continues on next page)

(continued from previous page)

```
         <activeAccounts> ACCTS </activeAccounts>
         <schedule> SCHED </schedule>
      requires ACCTFROM =/=K ACCTTO
       andBool notBool ACCTTO in ACCTS
       andBool (VALUE >Int 0 orBool notBool Gemptyisnonexistent << SCHED >>)

    rule <k> #transferFunds ACCTFROM ACCTTO 0 => . ... </k>
         <activeAccounts> ACCTS </activeAccounts>
         <schedule> SCHED </schedule>
      requires ACCTFROM =/=K ACCTTO
       andBool notBool ACCTTO in ACCTS
       andBool Gemptyisnonexistent << SCHED >>
```

4.3.2 Invalid Operator

We use `INVALID` both for marking the designated invalid operator, and `UNDEFINED(_)` for garbage bytes in the input program.

```
    syntax InvalidOp ::= "INVALID" | "UNDEFINED" "(" Int ")"
 // ---------------------------------------------------------
```

4.3.3 Stack Manipulations

Some operators don't calculate anything, they just push the stack around a bit.

```
    syntax UnStackOp ::= "POP"
 // --------------------------
    rule <k> POP W => . ... </k>

    syntax StackOp ::= DUP ( Int ) | SWAP ( Int )
 // ---------------------------------------------
    rule <k> DUP(N)  WS:WordStack => #setStack ((WS [ N -Int 1 ]) : WS)                  ␣
↪        ... </k>
    rule <k> SWAP(N) (W0 : WS)    => #setStack ((WS [ N -Int 1 ]) : (WS [ N -Int 1 :=␣
↪W0 ])) ... </k>

    syntax PushOp ::= PUSH ( Int , Int )
 // ------------------------------------
    rule <k> PUSH(_, W) => W ~> #push ... </k>
```

4.3.4 Local Memory

These operations are getters/setters of the local execution memory.

```
    syntax UnStackOp ::= "MLOAD"
 // ----------------------------
    rule <k> MLOAD INDEX => #asWord(#range(LM, INDEX, 32)) ~> #push ... </k>
         <localMem> LM </localMem>

    syntax BinStackOp ::= "MSTORE" | "MSTORE8"
 // ------------------------------------------
```

(continues on next page)

(continued from previous page)

```
    rule <k> MSTORE INDEX VALUE => . ... </k>
         <localMem> LM => LM [ INDEX := #padToWidth(32, #asByteStack(VALUE)) ] </
↪localMem>

    rule <k> MSTORE8 INDEX VALUE => . ... </k>
         <localMem> LM => LM [ INDEX <- (VALUE modInt 256) ] </localMem>
```

4.3.5 Expressions

Expression calculations are simple and don't require anything but the arguments from the `wordStack` to operate.

NOTE: We have to call the opcode `OR` by `EVMOR` instead, because K has trouble parsing it/compiling the definition otherwise.

```
    syntax UnStackOp ::= "ISZERO" | "NOT"
 // -------------------------------------
    rule <k> ISZERO W => W ==Word 0 ~> #push ... </k>
    rule <k> NOT    W => ~Word W    ~> #push ... </k>

    syntax BinStackOp ::= "ADD" | "MUL" | "SUB" | "DIV" | "EXP" | "MOD"
 // -------------------------------------------------------------------
    rule <k> ADD W0 W1 => W0 +Word W1 ~> #push ... </k>
    rule <k> MUL W0 W1 => W0 *Word W1 ~> #push ... </k>
    rule <k> SUB W0 W1 => W0 -Word W1 ~> #push ... </k>
    rule <k> DIV W0 W1 => W0 /Word W1 ~> #push ... </k>
    rule <k> EXP W0 W1 => W0 ^Word W1 ~> #push ... </k>
    rule <k> MOD W0 W1 => W0 %Word W1 ~> #push ... </k>

    syntax BinStackOp ::= "SDIV" | "SMOD"
 // -------------------------------------
    rule <k> SDIV W0 W1 => W0 /sWord W1 ~> #push ... </k>
    rule <k> SMOD W0 W1 => W0 %sWord W1 ~> #push ... </k>

    syntax TernStackOp ::= "ADDMOD" | "MULMOD"
 // ------------------------------------------
    rule <k> ADDMOD W0 W1 W2 => (W0 +Int W1) %Word W2 ~> #push ... </k>
    rule <k> MULMOD W0 W1 W2 => (W0 *Int W1) %Word W2 ~> #push ... </k>

    syntax BinStackOp ::= "BYTE" | "SIGNEXTEND"
 // -------------------------------------------
    rule <k> BYTE INDEX W     => byte(INDEX, W)     ~> #push ... </k>
    rule <k> SIGNEXTEND W0 W1 => signextend(W0, W1) ~> #push ... </k>

    syntax BinStackOp ::= "AND" | "EVMOR" | "XOR"
 // ---------------------------------------------
    rule <k> AND   W0 W1 => W0 &Word W1   ~> #push ... </k>
    rule <k> EVMOR W0 W1 => W0 |Word W1   ~> #push ... </k>
    rule <k> XOR   W0 W1 => W0 xorWord W1 ~> #push ... </k>

    syntax BinStackOp ::= "LT" | "GT" | "EQ"
 // ----------------------------------------
    rule <k> LT W0 W1 => W0 <Word  W1 ~> #push ... </k>
    rule <k> GT W0 W1 => W0 >Word  W1 ~> #push ... </k>
    rule <k> EQ W0 W1 => W0 ==Word W1 ~> #push ... </k>
```

(continues on next page)

(continued from previous page)

```
    syntax BinStackOp ::= "SLT" | "SGT"
 // -----------------------------------
    rule <k> SLT W0 W1 => W0 s<Word W1 ~> #push ... </k>
    rule <k> SGT W0 W1 => W1 s<Word W0 ~> #push ... </k>

    syntax BinStackOp ::= "SHA3"
 // ----------------------------
    rule <k> SHA3 MEMSTART MEMWIDTH => keccak(#range(LM, MEMSTART, MEMWIDTH)) ~>
↪#push ... </k>
         <localMem> LM </localMem>
```

4.3.6 Local State

These operators make queries about the current execution state.

```
    syntax NullStackOp ::= "PC" | "GAS" | "GASPRICE" | "GASLIMIT"
 // -------------------------------------------------------------
    rule <k> PC       => PCOUNT ~> #push ... </k> <pc> PCOUNT </pc>
    rule <k> GAS      => GAVAIL ~> #push ... </k> <gas> GAVAIL </gas>
    rule <k> GASPRICE => GPRICE ~> #push ... </k> <gasPrice> GPRICE </gasPrice>
    rule <k> GASLIMIT => GLIMIT ~> #push ... </k> <gasLimit> GLIMIT </gasLimit>

    syntax NullStackOp ::= "COINBASE" | "TIMESTAMP" | "NUMBER" | "DIFFICULTY"
 // -------------------------------------------------------------------------
    rule <k> COINBASE   => CB   ~> #push ... </k> <coinbase> CB </coinbase>
    rule <k> TIMESTAMP  => TS   ~> #push ... </k> <timestamp> TS </timestamp>
    rule <k> NUMBER     => NUMB ~> #push ... </k> <number> NUMB </number>
    rule <k> DIFFICULTY => DIFF ~> #push ... </k> <difficulty> DIFF </difficulty>

    syntax NullStackOp ::= "ADDRESS" | "ORIGIN" | "CALLER" | "CALLVALUE"
 // --------------------------------------------------------------------
    rule <k> ADDRESS   => ACCT ~> #push ... </k> <id> ACCT </id>
    rule <k> ORIGIN    => ORG  ~> #push ... </k> <origin> ORG </origin>
    rule <k> CALLER    => CL   ~> #push ... </k> <caller> CL </caller>
    rule <k> CALLVALUE => CV   ~> #push ... </k> <callValue> CV </callValue>

    syntax NullStackOp ::= "MSIZE" | "CODESIZE"
 // -------------------------------------------
    rule <k> MSIZE    => 32 *Word MU          ~> #push ... </k> <memoryUsed> MU </
↪memoryUsed>
    rule <k> CODESIZE => #sizeWordStack(PGM) ~> #push ... </k> <programBytes> PGM </
↪programBytes>

    syntax TernStackOp ::= "CODECOPY"
 // ---------------------------------
    rule <k> CODECOPY MEMSTART PGMSTART WIDTH => . ... </k>
         <programBytes> PGM </programBytes>
         <localMem> LM => LM [ MEMSTART := PGM [ PGMSTART .. WIDTH ] ] </localMem>

    syntax UnStackOp ::= "BLOCKHASH"
 // --------------------------------
```

When running as a node, the blockhash will be retrieved from the running client. Otherwise, it is calculated here using the “shortcut” formula used for running tests.

```
    rule <k> BLOCKHASH N => #blockhash(HASHES, N, HI -Int 1, 0) ~> #push ... </k>
         <number>    HI      </number>
         <blockhash> HASHES </blockhash>

    syntax Int ::= #blockhash ( List , Int , Int , Int ) [function]
 // ---------------------------------------------------------------
    rule #blockhash(_, N, HI, _) => 0 requires N >Int HI
    rule #blockhash(_, _, _, 256) => 0
    rule #blockhash(ListItem(0) _, _, _, _) => 0
    rule #blockhash(ListItem(H) _, N, N, _) => H
    rule #blockhash(ListItem(_) L, N, HI, A) => #blockhash(L, N, HI -Int 1, A +Int 1)␣
↪[owise]
```

4.4 EVM OpCodes

4.4.1 EVM Control Flow

The `JUMP*` family of operations affect the current program counter.

```
    syntax NullStackOp ::= "JUMPDEST"
 // ---------------------------------
    rule <k> JUMPDEST => . ... </k>

    syntax UnStackOp ::= "JUMP"
 // ---------------------------
    rule <k> JUMP DEST => . ... </k>
         <pc> _ => DEST </pc>
         <program> ... DEST |-> JUMPDEST ... </program>

    rule <k> JUMP DEST => #end EVMC_BAD_JUMP_DESTINATION ... </k>
         <program> ... DEST |-> OP ... </program>
      requires OP =/=K JUMPDEST

    rule <k> JUMP DEST => #end EVMC_BAD_JUMP_DESTINATION ... </k>
         <program> PGM </program>
      requires notBool (DEST in_keys(PGM))

    syntax BinStackOp ::= "JUMPI"
 // -----------------------------
    rule <k> JUMPI DEST I => . ... </k>
         <pc> PCOUNT => PCOUNT +Int 1 </pc>
      requires I ==Int 0

    rule <k> JUMPI DEST I => JUMP DEST ... </k>
      requires I =/=Int 0
```

4.4.2 STOP, REVERT, and RETURN

```
    syntax NullStackOp ::= "STOP"
 // -----------------------------
    rule <k> STOP => #end EVMC_SUCCESS ... </k>
         <output> _ => .WordStack </output>
```

(continues on next page)

(continued from previous page)

```
    syntax BinStackOp ::= "RETURN"
 // ------------------------------
    rule <k> RETURN RETSTART RETWIDTH => #end EVMC_SUCCESS ... </k>
         <output> _ => #range(LM, RETSTART, RETWIDTH) </output>
         <localMem> LM </localMem>

    syntax BinStackOp ::= "REVERT"
 // ------------------------------
    rule <k> REVERT RETSTART RETWIDTH => #end EVMC_REVERT ... </k>
         <output> _ => #range(LM, RETSTART, RETWIDTH) </output>
         <localMem> LM </localMem>
```

4.4.3 Call Data

These operators query about the current `CALL*` state.

```
    syntax NullStackOp ::= "CALLDATASIZE"
 // -------------------------------------
    rule <k> CALLDATASIZE => #sizeWordStack(CD) ~> #push ... </k>
         <callData> CD </callData>

    syntax UnStackOp ::= "CALLDATALOAD"
 // -----------------------------------
    rule <k> CALLDATALOAD DATASTART => #asWord(CD [ DATASTART .. 32 ]) ~> #push ... </
↪k>
         <callData> CD </callData>

    syntax TernStackOp ::= "CALLDATACOPY"
 // -------------------------------------
    rule <k> CALLDATACOPY MEMSTART DATASTART DATAWIDTH => . ... </k>
         <localMem> LM => LM [ MEMSTART := CD [ DATASTART .. DATAWIDTH ] ] </localMem>
         <callData> CD </callData>
```

4.4.4 Return Data

These operators query about the current return data buffer.

```
    syntax NullStackOp ::= "RETURNDATASIZE"
 // ---------------------------------------
    rule <k> RETURNDATASIZE => #sizeWordStack(RD) ~> #push ... </k>
         <output> RD </output>

    syntax TernStackOp ::= "RETURNDATACOPY"
 // ----------------------------------------
    rule <k> RETURNDATACOPY MEMSTART DATASTART DATAWIDTH => . ... </k>
         <localMem> LM => LM [ MEMSTART := RD [ DATASTART .. DATAWIDTH ] ] </localMem>
         <output> RD </output>
      requires DATASTART +Int DATAWIDTH <=Int #sizeWordStack(RD)

    rule <k> RETURNDATACOPY MEMSTART DATASTART DATAWIDTH => #end EVMC_INVALID_MEMORY_
↪ACCESS ... </k>
         <output> RD </output>
      requires DATASTART +Int DATAWIDTH >Int #sizeWordStack(RD)
```

4.4.5 Log Operations

```
    syntax BinStackOp ::= LogOp
    syntax LogOp ::= LOG ( Int )
 // ----------------------------
    rule <k> LOG(N) MEMSTART MEMWIDTH => . ... </k>
         <id> ACCT </id>
         <wordStack> WS => #drop(N, WS) </wordStack>
         <localMem> LM </localMem>
         <log> ... (.List => ListItem({ ACCT | WordStack2List(#take(N, WS)) |
↪#range(LM, MEMSTART, MEMWIDTH) })) </log>
      requires #sizeWordStack(WS) >=Int N
```

4.5 Ethereum Network OpCodes

Operators that require access to the rest of the Ethereum network world-state can be taken as a first draft of a "blockchain generic" language.

4.5.1 Account Queries

TODO: It's unclear what to do in the case of an account not existing for these operators. `BALANCE` is specified to push 0 in this case, but the others are not specified. For now, I assume that they instantiate an empty account and use the empty data.

```
    syntax UnStackOp ::= "BALANCE"
 // ------------------------------
    rule <k> BALANCE ACCT => BAL ~> #push ... </k>
         <account>
           <acctID> ACCT </acctID>
           <balance> BAL </balance>
           ...
         </account>

    rule <k> BALANCE ACCT => 0 ~> #push ... </k>
         <activeAccounts> ACCTS </activeAccounts>
      requires notBool ACCT in ACCTS

    syntax UnStackOp ::= "EXTCODESIZE"
 // ----------------------------------
    rule <k> EXTCODESIZE ACCT => #sizeWordStack(CODE) ~> #push ... </k>
         <account>
           <acctID> ACCT </acctID>
           <code> CODE </code>
           ...
         </account>

    rule <k> EXTCODESIZE ACCT => 0 ~> #push ... </k>
         <activeAccounts> ACCTS </activeAccounts>
      requires notBool ACCT in ACCTS
```

TODO: What should happen in the case that the account doesn't exist with `EXTCODECOPY`? Should we pad zeros (for the copied "program")?

```
    syntax QuadStackOp ::= "EXTCODECOPY"
 // ------------------------------------
    rule <k> EXTCODECOPY ACCT MEMSTART PGMSTART WIDTH => . ... </k>
         <localMem> LM => LM [ MEMSTART := PGM [ PGMSTART .. WIDTH ] ] </localMem>
         <account>
           <acctID> ACCT </acctID>
           <code> PGM </code>
           ...
         </account>

    rule <k> EXTCODECOPY ACCT MEMSTART PGMSTART WIDTH => . ... </k>
         <activeAccounts> ACCTS </activeAccounts>
      requires notBool ACCT in ACCTS
```

4.5.2 Account Storage Operations

These rules reach into the network state and load/store from account storage:

```
    syntax UnStackOp ::= "SLOAD"
 // ----------------------------
    rule <k> SLOAD INDEX => 0 ~> #push ... </k>
         <id> ACCT </id>
         <account>
           <acctID> ACCT </acctID>
           <storage> STORAGE </storage>
           ...
         </account> requires notBool INDEX in_keys(STORAGE)

    rule <k> SLOAD INDEX => VALUE ~> #push ... </k>
         <id> ACCT </id>
         <account>
           <acctID> ACCT </acctID>
           <storage> ... INDEX |-> VALUE ... </storage>
           ...
         </account>

    syntax BinStackOp ::= "SSTORE"
 // ------------------------------
    rule <k> SSTORE INDEX VALUE => . ... </k>
         <id> ACCT </id>
         <account>
           <acctID> ACCT </acctID>
           <storage> ... (INDEX |-> (OLD => VALUE)) ... </storage>
           ...
         </account>
         <refund> R => #if OLD =/=Int 0 andBool VALUE ==Int 0
                         #then R +Word Rsstoreclear < SCHED >
                         #else R
                       #fi
         </refund>
         <schedule> SCHED </schedule>

    rule <k> SSTORE INDEX VALUE => . ... </k>
         <id> ACCT </id>
         <account>
           <acctID> ACCT </acctID>
```

(continues on next page)

(continued from previous page)

```
          <storage> STORAGE => STORAGE [ INDEX <- VALUE ] </storage>
          ...
        </account>
      requires notBool (INDEX in_keys(STORAGE))
```

4.5.3 Call Operations

The various CALL* (and other inter-contract control flow) operations will be desugared into these InternalOps.

- The callLog is used to store the CALL*/CREATE operations so that we can compare them against the test-set.

```
    syntax Call ::= "{" Int "|" Int "|" Int "|" WordStack "}"
 // -----------------------------------------------------------
```

- #call______ takes the calling account, the account to execute as, the account whose code should execute, the gas limit, the amount to transfer, and the arguments.
- #callWithCode_______ takes the calling account, the accout to execute as, the code to execute (as a map), the gas limit, the amount to transfer, and the arguments.
- #return__ is a placeholder for the calling program, specifying where to place the returned data in memory.

```
    syntax InternalOp ::= "#checkCall" Int Int
                        | "#call" Int Int Int Exp Int Int WordStack Bool [strict(4)]
                        | "#callWithCode" Int Int Map WordStack Int Int Int WordStack␣
↪Bool
                        | "#mkCall" Int Int Map WordStack Int Int Int WordStack Bool
 // ------------------------------------------------------------------------------------
    rule <k> #checkCall ACCT VALUE ~> #call _ _ _ GLIMIT _ _ _ _
          => #refund GLIMIT ~> #pushCallStack ~> #pushWorldState
          ~> #end #if VALUE >Int BAL #then EVMC_BALANCE_UNDERFLOW #else EVMC_CALL_
↪DEPTH_EXCEEDED #fi
          ...
         </k>
         <callDepth> CD </callDepth>
         <output> _ => .WordStack </output>
         <account>
           <acctID> ACCT </acctID>
           <balance> BAL </balance>
           ...
         </account>
      requires VALUE >Int BAL orBool CD >=Int 1024

     rule <k> #checkCall ACCT VALUE => . ... </k>
         <callDepth> CD </callDepth>
         <account>
           <acctID> ACCT </acctID>
           <balance> BAL </balance>
           ...
         </account>
      requires notBool (VALUE >Int BAL orBool CD >=Int 1024)

    rule <k> #call ACCTFROM ACCTTO ACCTCODE GLIMIT:Int VALUE APPVALUE ARGS STATIC
          => #callWithCode ACCTFROM ACCTTO (0 |-> #precompiled(ACCTCODE)) .WordStack␣
↪GLIMIT VALUE APPVALUE ARGS STATIC
         ...
```

(continues on next page)

(continued from previous page)

```
         </k>
         <schedule> SCHED </schedule>
      requires ACCTCODE in #precompiledAccounts(SCHED)

    rule <k> #call ACCTFROM ACCTTO ACCTCODE GLIMIT:Int VALUE APPVALUE ARGS STATIC
          => #callWithCode ACCTFROM ACCTTO #asMapOpCodes(#dasmOpCodes(CODE, SCHED))␣
↪CODE GLIMIT VALUE APPVALUE ARGS STATIC
         ...
         </k>
         <schedule> SCHED </schedule>
         <acctID> ACCTCODE </acctID>
         <code> CODE </code>
      requires notBool ACCTCODE in #precompiledAccounts(SCHED)

    rule <k> #call ACCTFROM ACCTTO ACCTCODE GLIMIT:Int VALUE APPVALUE ARGS STATIC
          => #callWithCode ACCTFROM ACCTTO .Map .WordStack GLIMIT VALUE APPVALUE ARGS␣
↪STATIC
         ...
         </k>
         <activeAccounts> ACCTS </activeAccounts>
         <schedule> SCHED </schedule>
      requires notBool ACCTCODE in #precompiledAccounts(SCHED) andBool notBool␣
↪ACCTCODE in ACCTS

    rule <k> #callWithCode ACCTFROM ACCTTO CODE BYTES GLIMIT VALUE APPVALUE ARGS␣
↪STATIC
          => #pushCallStack ~> #pushWorldState
          ~> #transferFunds ACCTFROM ACCTTO VALUE
          ~> #mkCall ACCTFROM ACCTTO CODE BYTES GLIMIT VALUE APPVALUE ARGS STATIC
         ...
         </k>

    rule <k> #mkCall ACCTFROM ACCTTO CODE BYTES GLIMIT VALUE APPVALUE ARGS STATIC:Bool
          => #initVM ~> #execute
         ...
         </k>
         <callDepth> CD => CD +Int 1 </callDepth>
         <callData> _ => ARGS </callData>
         <callValue> _ => APPVALUE </callValue>
         <id> _ => ACCTTO </id>
         <gas> _ => GLIMIT </gas>
         <caller> _ => ACCTFROM </caller>
         <program> _ => CODE </program>
         <programBytes> _ => BYTES </programBytes>
         <static> OLDSTATIC:Bool => OLDSTATIC orBool STATIC </static>
         <touchedAccounts> ... .Set => SetItem(ACCTFROM) SetItem(ACCTTO) ... </
↪touchedAccounts>

    syntax KItem ::= "#initVM"
 // --------------------------
    rule <k> #initVM    => . ...       </k>
         <pc>          _ => 0          </pc>
         <memoryUsed>  _ => 0          </memoryUsed>
         <output>      _ => .WordStack </output>
         <wordStack>   _ => .WordStack </wordStack>
         <localMem>    _ => .Map       </localMem>
```

(continues on next page)

(continued from previous page)

```
    syntax KItem ::= "#return" Int Int
 // ----------------------------------
    rule <statusCode> _:ExceptionalStatusCode </statusCode>
         <k> #halt ~> #return _ _
          => #popCallStack ~> #popWorldState ~> 0 ~> #push
         ...
         </k>
         <output> _ => .WordStack </output>

    rule <statusCode> EVMC_REVERT </statusCode>
         <k> #halt ~> #return RETSTART RETWIDTH
          => #popCallStack ~> #popWorldState
          ~> 0 ~> #push ~> #refund GAVAIL ~> #setLocalMem RETSTART RETWIDTH OUT
         ...
         </k>
         <output> OUT </output>
         <gas> GAVAIL </gas>

    rule <statusCode> EVMC_SUCCESS </statusCode>
         <k> #halt ~> #return RETSTART RETWIDTH
          => #popCallStack ~> #dropWorldState
          ~> 1 ~> #push ~> #refund GAVAIL ~> #setLocalMem RETSTART RETWIDTH OUT
         ...
         </k>
         <output> OUT </output>
         <gas> GAVAIL </gas>

    syntax InternalOp ::= "#refund" Exp [strict]
                        | "#setLocalMem" Int Int WordStack
 // ------------------------------------------------------
    rule <k> #refund G:Int => . ... </k> <gas> GAVAIL => GAVAIL +Int G </gas>

    rule <k> #setLocalMem START WIDTH WS => . ... </k>
         <localMem> LM => LM [ START := #take(minInt(WIDTH, #sizeWordStack(WS)), WS)␣
↪] </localMem>
```

4.6 Ethereum Network OpCodes

4.6.1 Call Operations

For each CALL* operation, we make a corresponding call to #call and a state-change to setup the custom parts of the calling environment.

```
    syntax CallOp ::= "CALL"
 // ------------------------
    rule <k> CALL GCAP ACCTTO VALUE ARGSTART ARGWIDTH RETSTART RETWIDTH
          => #checkCall ACCTFROM VALUE
          ~> #call ACCTFROM ACCTTO ACCTTO Ccallgas(SCHED, #accountNonexistent(ACCTTO),
↪ GCAP, GAVAIL, VALUE) VALUE VALUE #range(LM, ARGSTART, ARGWIDTH) false
          ~> #return RETSTART RETWIDTH
         ...
         </k>
         <schedule> SCHED </schedule>
```

(continues on next page)

(continued from previous page)

```
        <id> ACCTFROM </id>
        <localMem> LM </localMem>
        <previousGas> GAVAIL </previousGas>

   syntax CallOp ::= "CALLCODE"
// ----------------------------
   rule <k> CALLCODE GCAP ACCTTO VALUE ARGSTART ARGWIDTH RETSTART RETWIDTH
         => #checkCall ACCTFROM VALUE
         ~> #call ACCTFROM ACCTFROM ACCTTO Ccallgas(SCHED,
↪#accountNonexistent(ACCTFROM), GCAP, GAVAIL, VALUE) VALUE VALUE #range(LM, ARGSTART,
↪ ARGWIDTH) false
         ~> #return RETSTART RETWIDTH
        ...
        </k>
        <schedule> SCHED </schedule>
        <id> ACCTFROM </id>
        <localMem> LM </localMem>
        <previousGas> GAVAIL </previousGas>

   syntax CallSixOp ::= "DELEGATECALL"
// -----------------------------------
   rule <k> DELEGATECALL GCAP ACCTTO ARGSTART ARGWIDTH RETSTART RETWIDTH
         => #checkCall ACCTFROM 0
         ~> #call ACCTAPPFROM ACCTFROM ACCTTO Ccallgas(SCHED,
↪#accountNonexistent(ACCTFROM), GCAP, GAVAIL, 0) 0 VALUE #range(LM, ARGSTART,␣
↪ARGWIDTH) false
         ~> #return RETSTART RETWIDTH
        ...
        </k>
        <schedule> SCHED </schedule>
        <id> ACCTFROM </id>
        <caller> ACCTAPPFROM </caller>
        <callValue> VALUE </callValue>
        <localMem> LM </localMem>
        <previousGas> GAVAIL </previousGas>

   syntax CallSixOp ::= "STATICCALL"
// ---------------------------------
   rule <k> STATICCALL GCAP ACCTTO ARGSTART ARGWIDTH RETSTART RETWIDTH
         => #checkCall ACCTFROM 0
         ~> #call ACCTFROM ACCTTO ACCTTO Ccallgas(SCHED, #accountNonexistent(ACCTTO),
↪ GCAP, GAVAIL, 0) 0 0 #range(LM, ARGSTART, ARGWIDTH) true
         ~> #return RETSTART RETWIDTH
        ...
        </k>
        <schedule> SCHED </schedule>
        <id> ACCTFROM </id>
        <localMem> LM </localMem>
        <previousGas> GAVAIL </previousGas>
```

4.6.2 Account Creation/Deletion

- `#create_____` transfers the endowment to the new account and triggers the execution of the initialization code.
- `#codeDeposit_` checks the result of initialization code and whether the code deposit can be paid, indicating

an error if not.

```
    syntax InternalOp ::= "#create" Int Int Int Int WordStack
                        | "#mkCreate" Int Int WordStack Int Int
                        | "#checkCreate" Int Int
                        | "#incrementNonce" Int
 // ------------------------------------------
    rule <k> #checkCreate ACCT VALUE ~> #create _ _ GAVAIL _ _
          => #refund GAVAIL ~> #pushCallStack ~> #pushWorldState
          ~> #end #if VALUE >Int BAL #then EVMC_BALANCE_UNDERFLOW #else EVMC_CALL_
↪DEPTH_EXCEEDED #fi
         ...
         </k>
         <callDepth> CD </callDepth>
         <output> _ => .WordStack </output>
         <account>
           <acctID> ACCT </acctID>
           <balance> BAL </balance>
           ...
         </account>
      requires VALUE >Int BAL orBool CD >=Int 1024

    rule <k> #checkCreate ACCT VALUE => #incrementNonce ACCT ... </k>
         <callDepth> CD </callDepth>
         <account>
           <acctID> ACCT </acctID>
           <balance> BAL </balance>
           ...
         </account>
      requires notBool (VALUE >Int BAL orBool CD >=Int 1024)

    rule <k> #create ACCTFROM ACCTTO GAVAIL VALUE INITCODE
          => #pushCallStack ~> #pushWorldState
          ~> #newAccount ACCTTO
          ~> #transferFunds ACCTFROM ACCTTO VALUE
          ~> #mkCreate ACCTFROM ACCTTO INITCODE GAVAIL VALUE
         ...
         </k>

    rule <k> #mkCreate ACCTFROM ACCTTO INITCODE GAVAIL VALUE
          => #initVM ~> #execute
         ...
         </k>
         <schedule> SCHED </schedule>
         <id> ACCT => ACCTTO </id>
         <gas> OLDGAVAIL => GAVAIL </gas>
         <program> _ => #asMapOpCodes(#dasmOpCodes(INITCODE, SCHED)) </program>
         <programBytes> _ => INITCODE </programBytes>
         <caller> _ => ACCTFROM </caller>
         <callDepth> CD => CD +Int 1 </callDepth>
         <callData> _ => .WordStack </callData>
         <callValue> _ => VALUE </callValue>
         <account>
           <acctID> ACCTTO </acctID>
           <nonce> NONCE => #if Gemptyisnonexistent << SCHED >> #then NONCE +Int 1
↪#else NONCE #fi </nonce>
           ...
         </account>
```

(continues on next page)

(continued from previous page)

```
         <touchedAccounts> ... .Set => SetItem(ACCTFROM) SetItem(ACCTTO) ... </
↪touchedAccounts>

    rule <k> #incrementNonce ACCT => . ... </k>
         <account>
           <acctID> ACCT </acctID>
           <nonce> NONCE => NONCE +Int 1 </nonce>
           ...
         </account>

    syntax KItem ::= "#codeDeposit" Int
                   | "#mkCodeDeposit" Int
                   | "#finishCodeDeposit" Int WordStack
 // ---------------------------------------------------
    rule <statusCode> _:ExceptionalStatusCode </statusCode>
         <k> #halt ~> #codeDeposit _ => #popCallStack ~> #popWorldState ~> 0 ~> #push␣
↪... </k> <output> _ => .WordStack </output>
    rule <statusCode> EVMC_REVERT </statusCode>
         <k> #halt ~> #codeDeposit _ => #popCallStack ~> #popWorldState ~> #refund␣
↪GAVAIL ~> 0 ~> #push ... </k>
         <gas> GAVAIL </gas>

    rule <statusCode> EVMC_SUCCESS </statusCode>
         <k> #halt ~> #codeDeposit ACCT => #mkCodeDeposit ACCT ... </k>

    rule <k> #mkCodeDeposit ACCT
          => Gcodedeposit < SCHED > *Int #sizeWordStack(OUT) ~> #deductGas
          ~> #finishCodeDeposit ACCT OUT
         ...
         </k>
         <schedule> SCHED </schedule>
         <output> OUT => .WordStack </output>
      requires #sizeWordStack(OUT) <=Int maxCodeSize < SCHED >

    rule <k> #mkCodeDeposit ACCT => #popCallStack ~> #popWorldState ~> 0 ~> #push ...
↪</k>
         <schedule> SCHED </schedule>
         <output> OUT => .WordStack </output>
      requires #sizeWordStack(OUT) >Int maxCodeSize < SCHED >

    rule <k> #finishCodeDeposit ACCT OUT
          => #popCallStack ~> #dropWorldState
          ~> #refund GAVAIL ~> ACCT ~> #push
         ...
         </k>
         <gas> GAVAIL </gas>
         <account>
           <acctID> ACCT </acctID>
           <code> _ => OUT </code>
           ...
         </account>

    rule <statusCode> _:ExceptionalStatusCode </statusCode>
         <k> #halt ~> #finishCodeDeposit ACCT _
          => #popCallStack ~> #dropWorldState
          ~> #refund GAVAIL ~> ACCT ~> #push
         ...
```

(continues on next page)

(continued from previous page)

```
         </k>
         <gas> GAVAIL </gas>
         <schedule> FRONTIER </schedule>

    rule <statusCode> _:ExceptionalStatusCode </statusCode>
         <k> #halt ~> #finishCodeDeposit _ _ => #popCallStack ~> #popWorldState ~> 0␣
↪~> #push ... </k>
         <schedule> SCHED </schedule>
      requires SCHED =/=K FRONTIER
```

CREATE will attempt to #create the account using the initialization code and cleans up the result with #codeDeposit.

```
    syntax TernStackOp ::= "CREATE"
 // --------------------------------
    rule <k> CREATE VALUE MEMSTART MEMWIDTH
           => #checkCreate ACCT VALUE
           ~> #create ACCT #newAddr(ACCT, NONCE) #if Gstaticcalldepth << SCHED >>
↪#then GAVAIL #else #allBut64th(GAVAIL) #fi VALUE #range(LM, MEMSTART, MEMWIDTH)
           ~> #codeDeposit #newAddr(ACCT, NONCE)
          ...
         </k>
         <schedule> SCHED </schedule>
         <id> ACCT </id>
         <gas> GAVAIL => #if Gstaticcalldepth << SCHED >> #then 0 #else GAVAIL /Int␣
↪64 #fi </gas>
         <localMem> LM </localMem>
         <account>
           <acctID> ACCT </acctID>
           <nonce> NONCE </nonce>
           ...
         </account>
```

SELFDESTRUCT marks the current account for deletion and transfers funds out of the current account. Self destructing to yourself, unlike a regular transfer, destroys the balance in the account, irreparably losing it.

```
    syntax UnStackOp ::= "SELFDESTRUCT"
 // -----------------------------------
    rule <k> SELFDESTRUCT ACCTTO => #transferFunds ACCT ACCTTO BALFROM ~> #end EVMC_
↪SUCCESS ... </k>
         <schedule> SCHED </schedule>
         <id> ACCT </id>
         <selfDestruct> SDS (.Set => SetItem(ACCT)) </selfDestruct>
         <refund> RF => #if ACCT in SDS #then RF #else RF +Word Rselfdestruct < SCHED␣
↪> #fi </refund>
         <account>
           <acctID> ACCT </acctID>
           <balance> BALFROM </balance>
           ...
         </account>
         <output> _ => .WordStack </output>
         <touchedAccounts> ... .Set => SetItem(ACCT) SetItem(ACCTTO) ... </
↪touchedAccounts>
      requires ACCT =/=Int ACCTTO

    rule <k> SELFDESTRUCT ACCT => #end EVMC_SUCCESS ... </k>
```

(continues on next page)

(continued from previous page)

```
         <schedule> SCHED </schedule>
         <id> ACCT </id>
         <selfDestruct> SDS (.Set => SetItem(ACCT)) </selfDestruct>
         <refund> RF => #if ACCT in SDS #then RF #else RF +Word Rselfdestruct < SCHED␣
↪> #fi </refund>
         <account>
           <acctID> ACCT </acctID>
           <balance> BALFROM => 0 </balance>
           <nonce> NONCE </nonce>
           <code> CODE </code>
           ...
         </account>
         <output> _ => .WordStack </output>
         <touchedAccounts> ... .Set => SetItem(ACCT) ... </touchedAccounts>
```

4.7 Precompiled Contracts

- #precompiled is a placeholder for the 4 pre-compiled contracts at addresses 1 through 4.

```
    syntax NullStackOp   ::= PrecompiledOp
    syntax PrecompiledOp ::= #precompiled ( Int ) [function]
 // -------------------------------------------------------
    rule #precompiled(1) => ECREC
    rule #precompiled(2) => SHA256
    rule #precompiled(3) => RIP160
    rule #precompiled(4) => ID
    rule #precompiled(5) => MODEXP
    rule #precompiled(6) => ECADD
    rule #precompiled(7) => ECMUL
    rule #precompiled(8) => ECPAIRING

    syntax Set ::= #precompiledAccounts ( Schedule ) [function]
 // -----------------------------------------------------------
    rule #precompiledAccounts(DEFAULT)        => SetItem(1) SetItem(2) SetItem(3)␣
↪SetItem(4)
    rule #precompiledAccounts(FRONTIER)       => #precompiledAccounts(DEFAULT)
    rule #precompiledAccounts(HOMESTEAD)      => #precompiledAccounts(FRONTIER)
    rule #precompiledAccounts(EIP150)         => #precompiledAccounts(HOMESTEAD)
    rule #precompiledAccounts(EIP158)         => #precompiledAccounts(EIP150)
    rule #precompiledAccounts(BYZANTIUM)      => #precompiledAccounts(EIP158)␣
↪SetItem(5) SetItem(6) SetItem(7) SetItem(8)
    rule #precompiledAccounts(CONSTANTINOPLE) => #precompiledAccounts(BYZANTIUM)
```

- ECREC performs ECDSA public key recovery.
- SHA256 performs the SHA2-257 hash function.
- RIP160 performs the RIPEMD-160 hash function.
- ID is the identity function (copies input to output).

```
    syntax PrecompiledOp ::= "ECREC"
 // --------------------------------
    rule <k> ECREC => #end EVMC_SUCCESS ... </k>
         <callData> DATA </callData>
```

(continues on next page)

(continued from previous page)

```
         <output> _ => #ecrec(#sender(#unparseByteStack(DATA [ 0 .. 32 ]),
↪#asWord(DATA [ 32 .. 32 ]), #unparseByteStack(DATA [ 64 .. 32 ]),
↪#unparseByteStack(DATA [ 96 .. 32 ]))) </output>

    syntax WordStack ::= #ecrec ( Account ) [function]
 // --------------------------------------------------
    rule #ecrec(.Account) => .WordStack
    rule #ecrec(N:Int)    => #padToWidth(32, #asByteStack(N))

    syntax PrecompiledOp ::= "SHA256"
 // ---------------------------------
    rule <k> SHA256 => #end EVMC_SUCCESS ... </k>
         <callData> DATA </callData>
         <output> _ => #parseHexBytes(Sha256(#unparseByteStack(DATA))) </output>

    syntax PrecompiledOp ::= "RIP160"
 // ---------------------------------
    rule <k> RIP160 => #end EVMC_SUCCESS ... </k>
         <callData> DATA </callData>
         <output> _ => #padToWidth(32, #parseHexBytes(RipEmd160(
↪#unparseByteStack(DATA)))) </output>

    syntax PrecompiledOp ::= "ID"
 // -----------------------------
    rule <k> ID => #end EVMC_SUCCESS ... </k>
         <callData> DATA </callData>
         <output> _ => DATA </output>

    syntax PrecompiledOp ::= "MODEXP"
 // ---------------------------------
    rule <k> MODEXP => #end EVMC_SUCCESS ... </k>
         <callData> DATA </callData>
         <output> _ => #modexp1(#asWord(DATA [ 0 .. 32 ]), #asWord(DATA [ 32 .. 32 ]),
↪ #asWord(DATA [ 64 .. 32 ]), #drop(96,DATA)) </output>

    syntax WordStack ::= #modexp1 ( Int , Int , Int , WordStack ) [function]
                       | #modexp2 ( Int , Int , Int , WordStack ) [function]
                       | #modexp3 ( Int , Int , Int , WordStack ) [function]
                       | #modexp4 ( Int , Int , Int )             [function]
 // -------------------------------------------------------------------------
    rule #modexp1(BASELEN, EXPLEN,   MODLEN, DATA) => #modexp2(#asInteger(DATA [ 0 ..␣
↪BASELEN ]), EXPLEN, MODLEN, #drop(BASELEN, DATA)) requires MODLEN =/=Int 0
    rule #modexp1(_,       _,        0,      _)    => .WordStack
    rule #modexp2(BASE,    EXPLEN,   MODLEN, DATA) => #modexp3(BASE, #asInteger(DATA␣
↪[ 0 .. EXPLEN ]), MODLEN, #drop(EXPLEN, DATA))
    rule #modexp3(BASE,    EXPONENT, MODLEN, DATA) => #padToWidth(MODLEN,
↪#modexp4(BASE, EXPONENT, #asInteger(DATA [ 0 .. MODLEN ])))
    rule #modexp4(BASE,    EXPONENT, MODULUS)      => #asByteStack(powmod(BASE,␣
↪EXPONENT, MODULUS))

    syntax PrecompiledOp ::= "ECADD"
 // --------------------------------
    rule <k> ECADD => #ecadd((#asWord(DATA [ 0 .. 32 ]), #asWord(DATA [ 32 .. 32 ])),␣
↪(#asWord(DATA [ 64 .. 32 ]), #asWord(DATA [ 96 .. 32 ]))) ... </k>
         <callData> DATA </callData>

    syntax InternalOp ::= #ecadd(G1Point, G1Point)
```

(continues on next page)

(continued from previous page)

```
 // ----------------------------------------------
    rule <k> #ecadd(P1, P2) => #end EVMC_PRECOMPILE_FAILURE ... </k>
      requires notBool isValidPoint(P1) orBool notBool isValidPoint(P2)
    rule <k> #ecadd(P1, P2) => #end EVMC_SUCCESS ... </k> <output> _ =>
↪#point(BN128Add(P1, P2)) </output>
      requires isValidPoint(P1) andBool isValidPoint(P2)

    syntax PrecompiledOp ::= "ECMUL"
 // --------------------------------
    rule <k> ECMUL => #ecmul((#asWord(DATA [ 0 .. 32 ]), #asWord(DATA [ 32 .. 32 ])),
↪#asWord(DATA [ 64 .. 32 ])) ... </k>
         <callData> DATA </callData>

    syntax InternalOp ::= #ecmul(G1Point, Int)
 // ------------------------------------------
    rule <k> #ecmul(P, S) => #end EVMC_PRECOMPILE_FAILURE ... </k>
      requires notBool isValidPoint(P)
    rule <k> #ecmul(P, S) => #end EVMC_SUCCESS ... </k> <output> _ =>
↪#point(BN128Mul(P, S)) </output>
      requires isValidPoint(P)

    syntax WordStack ::= #point ( G1Point ) [function]
 // --------------------------------------------------
    rule #point((X, Y)) => #padToWidth(32, #asByteStack(X)) ++ #padToWidth(32,
↪#asByteStack(Y))

    syntax PrecompiledOp ::= "ECPAIRING"
 // ------------------------------------
    rule <k> ECPAIRING => #ecpairing(.List, .List, 0, DATA, #sizeWordStack(DATA)) ...
↪</k>
         <callData> DATA </callData>
      requires #sizeWordStack(DATA) modInt 192 ==Int 0
    rule <k> ECPAIRING => #end EVMC_PRECOMPILE_FAILURE ... </k>
         <callData> DATA </callData>
      requires #sizeWordStack(DATA) modInt 192 =/=Int 0

    syntax InternalOp ::= #ecpairing(List, List, Int, WordStack, Int)
 // -----------------------------------------------------------------
    rule <k> (.K => #checkPoint) ~> #ecpairing((.List => ListItem((#asWord(DATA [ I␣
↪.. 32 ]), #asWord(DATA [ I +Int 32 .. 32 ])))) _, (.List => ListItem((#asWord(DATA␣
↪[ I +Int 96 .. 32 ]) x #asWord(DATA [ I +Int 64 .. 32 ]) , #asWord(DATA [ I +Int␣
↪160 .. 32 ]) x #asWord(DATA [ I +Int 128 .. 32 ])))) _, I => I +Int 192, DATA, LEN)␣
↪... </k>
      requires I =/=Int LEN
    rule <k> #ecpairing(A, B, LEN, _, LEN) => #end EVMC_SUCCESS ... </k>
         <output> _ => #padToWidth(32, #asByteStack(bool2Word(BN128AtePairing(A,␣
↪B)))) </output>

    syntax InternalOp ::= "#checkPoint"
 // -----------------------------------
    rule <k> (#checkPoint => .) ~> #ecpairing(ListItem(AK::G1Point) _,␣
↪ListItem(BK::G2Point) _, _, _, _) ... </k>
      requires isValidPoint(AK) andBool isValidPoint(BK)
    rule <k> #checkPoint ~> #ecpairing(ListItem(AK::G1Point) _, ListItem(BK::G2Point)␣
↪_, _, _, _) => #end EVMC_PRECOMPILE_FAILURE ... </k>
      requires notBool isValidPoint(AK) orBool notBool isValidPoint(BK)
```

CHAPTER

FIVE

ETHEREUM GAS CALCULATION

5.1 Memory Consumption

Memory consumed is tracked to determine the appropriate amount of gas to charge for each operation. In the YellowPaper, each opcode is defined to consume zero gas unless specified otherwise next to the semantics of the opcode (appendix H).

- `#memory` computes the new memory size given the old size and next operator (with its arguments).
- `#memoryUsageUpdate` is the function `M` in appendix H of the YellowPaper which helps track the memory used.

```
    syntax Int ::= #memory ( OpCode , Int ) [function]
 // ------------------------------------------------
    rule #memory ( MLOAD INDEX     , MU ) => #memoryUsageUpdate(MU, INDEX, 32)
    rule #memory ( MSTORE INDEX _  , MU ) => #memoryUsageUpdate(MU, INDEX, 32)
    rule #memory ( MSTORE8 INDEX _ , MU ) => #memoryUsageUpdate(MU, INDEX, 1)

    rule #memory ( SHA3 START WIDTH   , MU ) => #memoryUsageUpdate(MU, START, WIDTH)
    rule #memory ( LOG(_) START WIDTH , MU ) => #memoryUsageUpdate(MU, START, WIDTH)

    rule #memory ( CODECOPY START _ WIDTH       , MU ) => #memoryUsageUpdate(MU,␣
↪START, WIDTH)
    rule #memory ( EXTCODECOPY _ START _ WIDTH  , MU ) => #memoryUsageUpdate(MU,␣
↪START, WIDTH)
    rule #memory ( CALLDATACOPY START _ WIDTH   , MU ) => #memoryUsageUpdate(MU,␣
↪START, WIDTH)
    rule #memory ( RETURNDATACOPY START _ WIDTH , MU ) => #memoryUsageUpdate(MU,␣
↪START, WIDTH)

    rule #memory ( CREATE _ START WIDTH , MU ) => #memoryUsageUpdate(MU, START, WIDTH)
    rule #memory ( RETURN START WIDTH   , MU ) => #memoryUsageUpdate(MU, START, WIDTH)
    rule #memory ( REVERT START WIDTH   , MU ) => #memoryUsageUpdate(MU, START, WIDTH)

    rule #memory ( COP:CallOp     _ _ _ ARGSTART ARGWIDTH RETSTART RETWIDTH , MU ) =>
↪#memoryUsageUpdate(#memoryUsageUpdate(MU, ARGSTART, ARGWIDTH), RETSTART, RETWIDTH)
    rule #memory ( CSOP:CallSixOp _ _   ARGSTART ARGWIDTH RETSTART RETWIDTH , MU ) =>
↪#memoryUsageUpdate(#memoryUsageUpdate(MU, ARGSTART, ARGWIDTH), RETSTART, RETWIDTH)
```

Grumble grumble, K sucks at `owise`.

```
    rule #memory(JUMP _,     MU) => MU
    rule #memory(JUMPI _ _,  MU) => MU
    rule #memory(JUMPDEST,   MU) => MU
```

(continues on next page)

(continued from previous page)

```
    rule #memory(SSTORE _ _,    MU) => MU
    rule #memory(SLOAD _,       MU) => MU

    rule #memory(ADD _ _,         MU) => MU
    rule #memory(SUB _ _,         MU) => MU
    rule #memory(MUL _ _,         MU) => MU
    rule #memory(DIV _ _,         MU) => MU
    rule #memory(EXP _ _,         MU) => MU
    rule #memory(MOD _ _,         MU) => MU
    rule #memory(SDIV _ _,        MU) => MU
    rule #memory(SMOD _ _,        MU) => MU
    rule #memory(SIGNEXTEND _ _, MU) => MU
    rule #memory(ADDMOD _ _ _,    MU) => MU
    rule #memory(MULMOD _ _ _,    MU) => MU

    rule #memory(NOT _,      MU) => MU
    rule #memory(AND _ _,    MU) => MU
    rule #memory(EVMOR _ _, MU) => MU
    rule #memory(XOR _ _,    MU) => MU
    rule #memory(BYTE _ _,   MU) => MU
    rule #memory(ISZERO _,   MU) => MU

    rule #memory(LT _ _,          MU) => MU
    rule #memory(GT _ _,          MU) => MU
    rule #memory(SLT _ _,         MU) => MU
    rule #memory(SGT _ _,         MU) => MU
    rule #memory(EQ _ _,          MU) => MU

    rule #memory(POP _,       MU) => MU
    rule #memory(PUSH(_, _), MU) => MU
    rule #memory(DUP(_) _,    MU) => MU
    rule #memory(SWAP(_) _,   MU) => MU

    rule #memory(STOP,            MU) => MU
    rule #memory(ADDRESS,         MU) => MU
    rule #memory(ORIGIN,          MU) => MU
    rule #memory(CALLER,          MU) => MU
    rule #memory(CALLVALUE,       MU) => MU
    rule #memory(CALLDATASIZE,    MU) => MU
    rule #memory(RETURNDATASIZE, MU) => MU
    rule #memory(CODESIZE,        MU) => MU
    rule #memory(GASPRICE,        MU) => MU
    rule #memory(COINBASE,        MU) => MU
    rule #memory(TIMESTAMP,       MU) => MU
    rule #memory(NUMBER,          MU) => MU
    rule #memory(DIFFICULTY,      MU) => MU
    rule #memory(GASLIMIT,        MU) => MU
    rule #memory(PC,              MU) => MU
    rule #memory(MSIZE,           MU) => MU
    rule #memory(GAS,             MU) => MU

    rule #memory(SELFDESTRUCT _, MU) => MU
    rule #memory(CALLDATALOAD _, MU) => MU
    rule #memory(EXTCODESIZE _,  MU) => MU
    rule #memory(BALANCE _,      MU) => MU
    rule #memory(BLOCKHASH _,    MU) => MU
```

(continues on next page)

(continued from previous page)

```
    rule #memory(_:PrecompiledOp, MU) => MU

    syntax Int ::= #memoryUsageUpdate ( Int , Int , Int ) [function]
 // ----------------------------------------------------------------
    rule #memoryUsageUpdate(MU, START, 0)      => MU
    rule #memoryUsageUpdate(MU, START, WIDTH) => maxInt(MU, (START +Int WIDTH) up/Int␣
↪32) requires WIDTH >Int 0
```

5.2 Execution Gas

The intrinsic gas calculation mirrors the style of the YellowPaper (appendix H).

- #gasExec loads all the relevant surrounding state and uses that to compute the intrinsic execution gas of each opcode.

```
    syntax InternalOp ::= #gasExec ( Schedule , OpCode )
 // ----------------------------------------------------
    rule <k> #gasExec(SCHED, SSTORE INDEX VALUE) => Csstore(SCHED, VALUE,
↪#lookup(STORAGE, INDEX)) ... </k>
         <id> ACCT </id>
         <account>
           <acctID> ACCT </acctID>
           <storage> STORAGE </storage>
           ...
         </account>

    rule <k> #gasExec(SCHED, EXP W0 0)  => Gexp < SCHED > ... </k>
    rule <k> #gasExec(SCHED, EXP W0 W1) => Gexp < SCHED > +Int (Gexpbyte < SCHED >␣
↪*Int (1 +Int (log256Int(W1)))) ... </k> requires W1 =/=K 0

    rule <k> #gasExec(SCHED, CALLDATACOPY    _ _ WIDTH) => Gverylow     < SCHED >␣
↪+Int (Gcopy < SCHED > *Int (WIDTH up/Int 32)) ... </k>
    rule <k> #gasExec(SCHED, RETURNDATACOPY  _ _ WIDTH) => Gverylow     < SCHED >␣
↪+Int (Gcopy < SCHED > *Int (WIDTH up/Int 32)) ... </k>
    rule <k> #gasExec(SCHED, CODECOPY        _ _ WIDTH) => Gverylow     < SCHED >␣
↪+Int (Gcopy < SCHED > *Int (WIDTH up/Int 32)) ... </k>
    rule <k> #gasExec(SCHED, EXTCODECOPY   _ _ _ WIDTH) => Gextcodecopy < SCHED >␣
↪+Int (Gcopy < SCHED > *Int (WIDTH up/Int 32)) ... </k>

    rule <k> #gasExec(SCHED, LOG(N) _ WIDTH) => (Glog < SCHED > +Int (Glogdata <␣
↪SCHED > *Int WIDTH) +Int (N *Int Glogtopic < SCHED >)) ... </k>

    rule <k> #gasExec(SCHED, CALL GCAP ACCTTO VALUE _ _ _ _) => Ccall(SCHED,
↪#accountNonexistent(ACCTTO), GCAP, GAVAIL, VALUE) ... </k>
         <gas> GAVAIL </gas>

    rule <k> #gasExec(SCHED, CALLCODE GCAP _ VALUE _ _ _ _) => Ccall(SCHED,
↪#accountNonexistent(ACCTFROM), GCAP, GAVAIL, VALUE) ... </k>
         <id> ACCTFROM </id>
         <gas> GAVAIL </gas>

    rule <k> #gasExec(SCHED, DELEGATECALL GCAP _ _ _ _ _) => Ccall(SCHED,
↪#accountNonexistent(ACCTFROM), GCAP, GAVAIL, 0) ... </k>
```

(continues on next page)

(continued from previous page)

```
         <id> ACCTFROM </id>
         <gas> GAVAIL </gas>

    rule <k> #gasExec(SCHED, STATICCALL GCAP ACCTTO _ _ _ _) => Ccall(SCHED,
↪#accountNonexistent(ACCTTO), GCAP, GAVAIL, 0) ... </k>
         <gas> GAVAIL </gas>

    rule <k> #gasExec(SCHED, SELFDESTRUCT ACCTTO) => Cselfdestruct(SCHED,
↪#accountNonexistent(ACCTTO), BAL) ... </k>
         <id> ACCTFROM </id>
         <account>
           <acctID> ACCTFROM </acctID>
           <balance> BAL </balance>
           ...
         </account>

    rule <k> #gasExec(SCHED, CREATE _ _ _) => Gcreate < SCHED > ... </k>

    rule <k> #gasExec(SCHED, SHA3 _ WIDTH) => Gsha3 < SCHED > +Int (Gsha3word < SCHED␣
↪> *Int (WIDTH up/Int 32)) ... </k>

    rule <k> #gasExec(SCHED, JUMPDEST) => Gjumpdest < SCHED > ... </k>
    rule <k> #gasExec(SCHED, SLOAD _)  => Gsload    < SCHED > ... </k>

    // Wzero
    rule <k> #gasExec(SCHED, STOP)       => Gzero < SCHED > ... </k>
    rule <k> #gasExec(SCHED, RETURN _ _) => Gzero < SCHED > ... </k>
    rule <k> #gasExec(SCHED, REVERT _ _) => Gzero < SCHED > ... </k>

    // Wbase
    rule <k> #gasExec(SCHED, ADDRESS)        => Gbase < SCHED > ... </k>
    rule <k> #gasExec(SCHED, ORIGIN)         => Gbase < SCHED > ... </k>
    rule <k> #gasExec(SCHED, CALLER)         => Gbase < SCHED > ... </k>
    rule <k> #gasExec(SCHED, CALLVALUE)      => Gbase < SCHED > ... </k>
    rule <k> #gasExec(SCHED, CALLDATASIZE)   => Gbase < SCHED > ... </k>
    rule <k> #gasExec(SCHED, RETURNDATASIZE) => Gbase < SCHED > ... </k>
    rule <k> #gasExec(SCHED, CODESIZE)       => Gbase < SCHED > ... </k>
    rule <k> #gasExec(SCHED, GASPRICE)       => Gbase < SCHED > ... </k>
    rule <k> #gasExec(SCHED, COINBASE)       => Gbase < SCHED > ... </k>
    rule <k> #gasExec(SCHED, TIMESTAMP)      => Gbase < SCHED > ... </k>
    rule <k> #gasExec(SCHED, NUMBER)         => Gbase < SCHED > ... </k>
    rule <k> #gasExec(SCHED, DIFFICULTY)     => Gbase < SCHED > ... </k>
    rule <k> #gasExec(SCHED, GASLIMIT)       => Gbase < SCHED > ... </k>
    rule <k> #gasExec(SCHED, POP _)          => Gbase < SCHED > ... </k>
    rule <k> #gasExec(SCHED, PC)             => Gbase < SCHED > ... </k>
    rule <k> #gasExec(SCHED, MSIZE)          => Gbase < SCHED > ... </k>
    rule <k> #gasExec(SCHED, GAS)            => Gbase < SCHED > ... </k>

    // Wverylow
    rule <k> #gasExec(SCHED, ADD _ _)        => Gverylow < SCHED > ... </k>
    rule <k> #gasExec(SCHED, SUB _ _)        => Gverylow < SCHED > ... </k>
    rule <k> #gasExec(SCHED, NOT _)          => Gverylow < SCHED > ... </k>
    rule <k> #gasExec(SCHED, LT _ _)         => Gverylow < SCHED > ... </k>
    rule <k> #gasExec(SCHED, GT _ _)         => Gverylow < SCHED > ... </k>
    rule <k> #gasExec(SCHED, SLT _ _)        => Gverylow < SCHED > ... </k>
    rule <k> #gasExec(SCHED, SGT _ _)        => Gverylow < SCHED > ... </k>
    rule <k> #gasExec(SCHED, EQ _ _)         => Gverylow < SCHED > ... </k>
```

(continues on next page)

(continued from previous page)

```
    rule <k> #gasExec(SCHED, ISZERO _)        => Gverylow < SCHED > ... </k>
    rule <k> #gasExec(SCHED, AND _ _)         => Gverylow < SCHED > ... </k>
    rule <k> #gasExec(SCHED, EVMOR _ _)       => Gverylow < SCHED > ... </k>
    rule <k> #gasExec(SCHED, XOR _ _)         => Gverylow < SCHED > ... </k>
    rule <k> #gasExec(SCHED, BYTE _ _)        => Gverylow < SCHED > ... </k>
    rule <k> #gasExec(SCHED, CALLDATALOAD _) => Gverylow < SCHED > ... </k>
    rule <k> #gasExec(SCHED, MLOAD _)         => Gverylow < SCHED > ... </k>
    rule <k> #gasExec(SCHED, MSTORE _ _)      => Gverylow < SCHED > ... </k>
    rule <k> #gasExec(SCHED, MSTORE8 _ _)     => Gverylow < SCHED > ... </k>
    rule <k> #gasExec(SCHED, PUSH(_, _))      => Gverylow < SCHED > ... </k>
    rule <k> #gasExec(SCHED, DUP(_) _)        => Gverylow < SCHED > ... </k>
    rule <k> #gasExec(SCHED, SWAP(_) _)       => Gverylow < SCHED > ... </k>

    // Wlow
    rule <k> #gasExec(SCHED, MUL _ _)         => Glow < SCHED > ... </k>
    rule <k> #gasExec(SCHED, DIV _ _)         => Glow < SCHED > ... </k>
    rule <k> #gasExec(SCHED, SDIV _ _)        => Glow < SCHED > ... </k>
    rule <k> #gasExec(SCHED, MOD _ _)         => Glow < SCHED > ... </k>
    rule <k> #gasExec(SCHED, SMOD _ _)        => Glow < SCHED > ... </k>
    rule <k> #gasExec(SCHED, SIGNEXTEND _ _) => Glow < SCHED > ... </k>

    // Wmid
    rule <k> #gasExec(SCHED, ADDMOD _ _ _) => Gmid < SCHED > ... </k>
    rule <k> #gasExec(SCHED, MULMOD _ _ _) => Gmid < SCHED > ... </k>
    rule <k> #gasExec(SCHED, JUMP _) => Gmid < SCHED > ... </k>

    // Whigh
    rule <k> #gasExec(SCHED, JUMPI _ _) => Ghigh < SCHED > ... </k>

    rule <k> #gasExec(SCHED, EXTCODESIZE _) => Gextcodesize < SCHED > ... </k>
    rule <k> #gasExec(SCHED, BALANCE _)     => Gbalance     < SCHED > ... </k>
    rule <k> #gasExec(SCHED, BLOCKHASH _)   => Gblockhash   < SCHED > ... </k>

    // Precompiled
    rule <k> #gasExec(_, ECREC)  => 3000 ... </k>
    rule <k> #gasExec(_, SHA256) =>   60 +Int  12 *Int (#sizeWordStack(DATA) up/Int␣
↪32) ... </k> <callData> DATA </callData>
    rule <k> #gasExec(_, RIP160) => 600 +Int 120 *Int (#sizeWordStack(DATA) up/Int␣
↪32) ... </k> <callData> DATA </callData>
    rule <k> #gasExec(_, ID)     =>   15 +Int   3 *Int (#sizeWordStack(DATA) up/Int␣
↪32) ... </k> <callData> DATA </callData>
    rule <k> #gasExec(_, MODEXP)
          => #multComplexity(maxInt(#asWord(DATA [ 0 .. 32 ]), #asWord(DATA [ 64 ..␣
↪32 ]))) *Int maxInt(#adjustedExpLength(#asWord(DATA [ 0 .. 32 ]), #asWord(DATA [ 32␣
↪.. 32 ]), DATA), 1) /Int Gquaddivisor < SCHED >
         ...
         </k>
         <schedule> SCHED </schedule>
         <callData> DATA </callData>

    rule <k> #gasExec(_, ECADD)     => 500   ... </k>
    rule <k> #gasExec(_, ECMUL)     => 40000 ... </k>
    rule <k> #gasExec(_, ECPAIRING) => 100000 +Int (#sizeWordStack(DATA) /Int 192)␣
↪*Int 80000 ... </k> <callData> DATA </callData>
```

There are several helpers for calculating gas (most of them also specified in the YellowPaper).

```
    syntax Exp     ::= Int
    syntax KResult ::= Int
    syntax Exp ::= Ccall          ( Schedule , BExp , Int , Int , Int ) [strict(2)]
                 | Ccallgas       ( Schedule , BExp , Int , Int , Int ) [strict(2)]
                 | Cselfdestruct  ( Schedule , BExp , Int )             [strict(2)]
 // ------------------------------------------------------------------------------
    rule <k> Ccall(SCHED, ISEMPTY:Bool, GCAP, GAVAIL, VALUE)
          => Cextra(SCHED, VALUE, ISEMPTY) +Int Cgascap(SCHED, GCAP, GAVAIL,␣
↪Cextra(SCHED, VALUE, ISEMPTY)) ... </k>

    rule <k> Ccallgas(SCHED, ISEMPTY:Bool, GCAP, GAVAIL, VALUE)
          => Cgascap(SCHED, GCAP, GAVAIL, Cextra(SCHED, VALUE, ISEMPTY)) +Int #if␣
↪VALUE ==Int 0 #then 0 #else Gcallstipend < SCHED > #fi ... </k>

    rule <k> Cselfdestruct(SCHED, ISEMPTY:Bool, BAL)
          => Gselfdestruct < SCHED > +Int Cnew(SCHED, BAL, ISEMPTY andBool␣
↪Gselfdestructnewaccount << SCHED >>) ... </k>

    syntax Int ::= Cgascap ( Schedule , Int , Int , Int ) [function]
                 | Csstore ( Schedule , Int , Int )       [function]
                 | Cextra  ( Schedule , Int , Bool )      [function]
                 | Cnew    ( Schedule , Int , Bool )      [function]
                 | Cxfer   ( Schedule , Int )             [function]
 // ----------------------------------------------------------------
    rule Cgascap(SCHED, GCAP, GAVAIL, GEXTRA)
      => #if GAVAIL <Int GEXTRA orBool Gstaticcalldepth << SCHED >> #then GCAP #else␣
↪minInt(#allBut64th(GAVAIL -Int GEXTRA), GCAP) #fi

    rule Csstore(SCHED, VALUE, OLD)
      => #if VALUE =/=Int 0 andBool OLD ==Int 0 #then Gsstoreset < SCHED > #else␣
↪Gsstorereset < SCHED > #fi

    rule Cextra(SCHED, VALUE, ISEMPTY)
      => Gcall < SCHED > +Int Cnew(SCHED, VALUE, ISEMPTY) +Int Cxfer(SCHED, VALUE)

    rule Cnew(SCHED, VALUE, ISEMPTY:Bool)
      => #if ISEMPTY andBool (VALUE =/=Int 0 orBool Gzerovaluenewaccountgas << SCHED␣
↪>>) #then Gnewaccount < SCHED > #else 0 #fi

    rule Cxfer(SCHED, 0) => 0
    rule Cxfer(SCHED, N) => Gcallvalue < SCHED > requires N =/=K 0

    syntax BExp    ::= Bool
    syntax KResult ::= Bool
    syntax BExp ::= #accountNonexistent ( Int )
 // -------------------------------------------
    rule <k> #accountNonexistent(ACCT) => true ... </k>
         <activeAccounts> ACCTS </activeAccounts>
      requires notBool ACCT in ACCTS

    rule <k> #accountNonexistent(ACCT) => #accountEmpty(CODE, NONCE, BAL) andBool␣
↪Gemptyisnonexistent << SCHED >> ... </k>
         <schedule> SCHED </schedule>
         <account>
           <acctID>  ACCT  </acctID>
           <balance> BAL   </balance>
           <nonce>   NONCE </nonce>
```

(continues on next page)

(continued from previous page)

```
           <code>     CODE   </code>
           ...
         </account>

    syntax Bool ::= #accountEmpty ( WordStack , Int , Int ) [function,␣
↪klabel(accountEmpty)]
 // -------------------------------------------------------------------------------
↪------
    rule #accountEmpty(CODE, NONCE, BAL) => CODE ==K .WordStack andBool NONCE ==Int 0␣
↪andBool BAL ==Int 0

    syntax Int ::= #allBut64th ( Int ) [function]
 // ---------------------------------------------
    rule #allBut64th(N) => N -Int (N /Int 64)

    syntax Int ::= G0 ( Schedule , WordStack , Bool ) [function]
 // ------------------------------------------------------------
    rule G0(SCHED, .WordStack, true)  => Gtxcreate     < SCHED >
    rule G0(SCHED, .WordStack, false) => Gtransaction < SCHED >

    rule G0(SCHED, 0 : REST, ISCREATE) => Gtxdatazero    < SCHED > +Int G0(SCHED,␣
↪REST, ISCREATE)
    rule G0(SCHED, N : REST, ISCREATE) => Gtxdatanonzero < SCHED > +Int G0(SCHED,␣
↪REST, ISCREATE) requires N =/=Int 0

    syntax Int ::= "G*" "(" Int "," Int "," Int ")" [function]
 // ----------------------------------------------------------
    rule G*(GAVAIL, GLIMIT, REFUND) => GAVAIL +Int minInt((GLIMIT -Int GAVAIL)/Int 2,␣
↪REFUND)

    syntax Int ::= #multComplexity(Int) [function]
 // ----------------------------------------------
    rule #multComplexity(X) => X *Int X                                  requires␣
↪X <=Int 64
    rule #multComplexity(X) => X *Int X /Int 4 +Int 96 *Int X -Int 3072     requires␣
↪X >Int 64 andBool X <=Int 1024
    rule #multComplexity(X) => X *Int X /Int 16 +Int 480 *Int X -Int 199680 requires␣
↪X >Int 1024

    syntax Int ::= #adjustedExpLength(Int, Int, WordStack) [function]
                 | #adjustedExpLength(Int)                  [function, klabel(
↪#adjustedExpLengthAux)]
 // -------------------------------------------------------------------------------
↪--------------
    rule #adjustedExpLength(BASELEN, EXPLEN, DATA) => #if EXPLEN <=Int 32 #then 0
↪#else 8 *Int (EXPLEN -Int 32) #fi +Int #adjustedExpLength(#asInteger(DATA [ 96 +Int␣
↪BASELEN .. minInt(EXPLEN, 32) ]))

    rule #adjustedExpLength(0) => 0
    rule #adjustedExpLength(1) => 0
    rule #adjustedExpLength(N) => 1 +Int #adjustedExpLength(N /Int 2) requires N >Int␣
↪1
```

5.3 Fee Schedule from C++ Implementation

5.3.1 From the C++ Implementation

The C++ Implementation of EVM specifies several different "profiles" for how the VM works. Here we provide each protocol from the C++ implementation, as the YellowPaper does not contain all the different profiles. Specify which profile by passing in the argument `-cSCHEDULE=<FEE_SCHEDULE>` when calling `krun` (the available `<FEE_SCHEDULE>` are supplied here).

A `ScheduleFlag` is a boolean determined by the fee schedule; applying a `ScheduleFlag` to a `Schedule` yields whether the flag is set or not.

```
    syntax Bool ::= ScheduleFlag "<<" Schedule ">>" [function]
 // ----------------------------------------------------------

    syntax ScheduleFlag ::= "Gselfdestructnewaccount" | "Gstaticcalldepth" |
↪"Gemptyisnonexistent" | "Gzerovaluenewaccountgas"
                          | "Ghasrevert"              | "Ghasreturndata"   |
↪"Ghasstaticcall"
 // --------------------------------------------------------------------------------
↪-------
```

5.3.2 Schedule Constants

A `ScheduleConst` is a constant determined by the fee schedule.

```
    syntax Int ::= ScheduleConst "<" Schedule ">" [function]
 // --------------------------------------------------------

    syntax ScheduleConst ::= "Gzero"         | "Gbase"          | "Gverylow"      |
↪"Glow"          | "Gmid"        | "Ghigh"
                          | "Gextcodesize" | "Gextcodecopy"   | "Gbalance"      |
↪"Gsload"        | "Gjumpdest"   | "Gsstoreset"
                          | "Gsstorereset" | "Rsstoreclear"   | "Rselfdestruct" |
↪"Gselfdestruct" | "Gcreate"     | "Gcodedeposit"   | "Gcall"
                          | "Gcallvalue"   | "Gcallstipend"   | "Gnewaccount"   |
↪"Gexp"          | "Gexpbyte"    | "Gmemory"        | "Gtxcreate"
                          | "Gtxdatazero"  | "Gtxdatanonzero" | "Gtransaction"  |
↪"Glog"          | "Glogdata"    | "Glogtopic"      | "Gsha3"
                          | "Gsha3word"    | "Gcopy"          | "Gblockhash"    |
↪"Gquadcoeff"    | "maxCodeSize" | "Rb"             | "Gquaddivisor"
 // --------------------------------------------------------------------------------
↪--------------------------------------------------------------
```

5.3.3 Default Schedule

```
    syntax Schedule ::= "DEFAULT"
 // -----------------------------
    rule Gzero    < DEFAULT > => 0
    rule Gbase    < DEFAULT > => 2
    rule Gverylow < DEFAULT > => 3
    rule Glow     < DEFAULT > => 5
    rule Gmid     < DEFAULT > => 8
```

(continues on next page)

(continued from previous page)

```
    rule Ghigh     < DEFAULT > => 10

    rule Gexp      < DEFAULT > => 10
    rule Gexpbyte  < DEFAULT > => 10
    rule Gsha3     < DEFAULT > => 30
    rule Gsha3word < DEFAULT > => 6

    rule Gsload       < DEFAULT > => 50
    rule Gsstoreset   < DEFAULT > => 20000
    rule Gsstorereset < DEFAULT > => 5000
    rule Rsstoreclear < DEFAULT > => 15000

    rule Glog      < DEFAULT > => 375
    rule Glogdata  < DEFAULT > => 8
    rule Glogtopic < DEFAULT > => 375

    rule Gcall        < DEFAULT > => 40
    rule Gcallstipend < DEFAULT > => 2300
    rule Gcallvalue   < DEFAULT > => 9000
    rule Gnewaccount  < DEFAULT > => 25000

    rule Gcreate       < DEFAULT > => 32000
    rule Gcodedeposit  < DEFAULT > => 200
    rule Gselfdestruct < DEFAULT > => 0
    rule Rselfdestruct < DEFAULT > => 24000

    rule Gmemory      < DEFAULT > => 3
    rule Gquadcoeff   < DEFAULT > => 512
    rule Gcopy        < DEFAULT > => 3
    rule Gquaddivisor < DEFAULT > => 20

    rule Gtransaction   < DEFAULT > => 21000
    rule Gtxcreate      < DEFAULT > => 53000
    rule Gtxdatazero    < DEFAULT > => 4
    rule Gtxdatanonzero < DEFAULT > => 68

    rule Gjumpdest    < DEFAULT > => 1
    rule Gbalance     < DEFAULT > => 20
    rule Gblockhash   < DEFAULT > => 20
    rule Gextcodesize < DEFAULT > => 20
    rule Gextcodecopy < DEFAULT > => 20

    rule maxCodeSize < DEFAULT > => 2 ^Int 32 -Int 1
    rule Rb          < DEFAULT > => 5 *Int (10 ^Int 18)

    rule Gselfdestructnewaccount << DEFAULT >> => false
    rule Gstaticcalldepth        << DEFAULT >> => true
    rule Gemptyisnonexistent     << DEFAULT >> => false
    rule Gzerovaluenewaccountgas << DEFAULT >> => true
    rule Ghasrevert              << DEFAULT >> => false
    rule Ghasreturndata          << DEFAULT >> => false
    rule Ghasstaticcall          << DEFAULT >> => false
```

```
struct EVMSchedule
{
    EVMSchedule(): tierStepGas(std::array<unsigned, 8>{{0, 2, 3, 5, 8, 10, 20, 0}}) {}
```

(continues on next page)

(continued from previous page)

```
    EVMSchedule(bool _efcd, bool _hdc, unsigned const& _txCreateGas):␣
↪exceptionalFailedCodeDeposit(_efcd), haveDelegateCall(_hdc), tierStepGas(std::array
↪<unsigned, 8>{{0, 2, 3, 5, 8, 10, 20, 0}}), txCreateGas(_txCreateGas) {}
    bool exceptionalFailedCodeDeposit = true;
    bool haveDelegateCall = true;
    bool eip150Mode = false;
    bool eip158Mode = false;
    bool haveRevert = false;
    bool haveReturnData = false;
    bool haveStaticCall = false;
    bool haveCreate2 = false;
    std::array<unsigned, 8> tierStepGas;

    unsigned expGas = 10;
    unsigned expByteGas = 10;
    unsigned sha3Gas = 30;
    unsigned sha3WordGas = 6;

    unsigned sloadGas = 50;
    unsigned sstoreSetGas = 20000;
    unsigned sstoreResetGas = 5000;
    unsigned sstoreRefundGas = 15000;

    unsigned logGas = 375;
    unsigned logDataGas = 8;
    unsigned logTopicGas = 375;

    unsigned callGas = 40;
    unsigned callStipend = 2300;
    unsigned callValueTransferGas = 9000;
    unsigned callNewAccountGas = 25000;

    unsigned createGas = 32000;
    unsigned createDataGas = 200;
    unsigned suicideGas = 0;
    unsigned suicideRefundGas = 24000;

    unsigned memoryGas = 3;
    unsigned quadCoeffDiv = 512;
    unsigned copyGas = 3;

    unsigned txGas = 21000;
    unsigned txCreateGas = 53000;
    unsigned txDataZeroGas = 4;
    unsigned txDataNonZeroGas = 68;

    unsigned jumpdestGas = 1;
    unsigned balanceGas = 20;
    unsigned blockhashGas = 20;
    unsigned extcodesizeGas = 20;
    unsigned extcodecopyGas = 20;

    unsigned maxCodeSize = unsigned(-1);

    bool staticCallDepthLimit() const { return !eip150Mode; }
    bool suicideNewAccountGas() const { return !eip150Mode; }
    bool suicideChargesNewAccountGas() const { return eip150Mode; }
```

(continues on next page)

(continued from previous page)

```
    bool emptinessIsNonexistence() const { return eip158Mode; }
    bool zeroValueTransferChargesNewAccountGas() const { return !eip158Mode; }
};
```

5.3.4 Frontier Schedule

```
    syntax Schedule ::= "FRONTIER"
 // ------------------------------
    rule Gtxcreate  < FRONTIER > => 21000
    rule SCHEDCONST < FRONTIER > => SCHEDCONST < DEFAULT > requires SCHEDCONST =/=K␣
↪Gtxcreate

    rule SCHEDFLAG << FRONTIER >> => SCHEDFLAG << DEFAULT >>
```

```
static const EVMSchedule FrontierSchedule = EVMSchedule(false, false, 21000);
```

5.3.5 Homestead Schedule

```
    syntax Schedule ::= "HOMESTEAD"
 // -------------------------------
    rule SCHEDCONST < HOMESTEAD > => SCHEDCONST < DEFAULT >

    rule SCHEDFLAG << HOMESTEAD >> => SCHEDFLAG << DEFAULT >>
```

```
static const EVMSchedule HomesteadSchedule = EVMSchedule(true, true, 53000);
```

5.3.6 EIP150 Schedule

```
    syntax Schedule ::= "EIP150"
 // ----------------------------
    rule Gbalance       < EIP150 > => 400
    rule Gsload         < EIP150 > => 200
    rule Gcall          < EIP150 > => 700
    rule Gselfdestruct  < EIP150 > => 5000
    rule Gextcodesize   < EIP150 > => 700
    rule Gextcodecopy   < EIP150 > => 700

    rule SCHEDCONST     < EIP150 > => SCHEDCONST < HOMESTEAD >
      requires notBool      ( SCHEDCONST ==K Gbalance      orBool SCHEDCONST ==K␣
↪Gsload       orBool SCHEDCONST ==K Gcall
                            orBool SCHEDCONST ==K Gselfdestruct orBool SCHEDCONST ==K␣
↪Gextcodesize orBool SCHEDCONST ==K Gextcodecopy
                            )

    rule Gselfdestructnewaccount << EIP150 >> => true
    rule Gstaticcalldepth        << EIP150 >> => false
    rule SCHEDCONST              << EIP150 >> => SCHEDCONST << HOMESTEAD >>
      requires notBool      ( SCHEDCONST ==K Gselfdestructnewaccount orBool␣
↪SCHEDCONST ==K Gstaticcalldepth )
```

```
static const EVMSchedule EIP150Schedule = []
{
    EVMSchedule schedule = HomesteadSchedule;
    schedule.eip150Mode = true;
    schedule.extcodesizeGas = 700;
    schedule.extcodecopyGas = 700;
    schedule.balanceGas = 400;
    schedule.sloadGas = 200;
    schedule.callGas = 700;
    schedule.suicideGas = 5000;
    return schedule;
}();
```

5.3.7 EIP158 Schedule

```
    syntax Schedule ::= "EIP158"
 // ----------------------------
    rule Gexpbyte    < EIP158 > => 50
    rule maxCodeSize < EIP158 > => 24576

    rule SCHEDCONST  < EIP158 > => SCHEDCONST < EIP150 > requires SCHEDCONST =/=K␣
↪Gexpbyte andBool SCHEDCONST =/=K maxCodeSize

    rule Gemptyisnonexistent      << EIP158 >> => true
    rule Gzerovaluenewaccountgas << EIP158 >> => false
    rule SCHEDCONST              << EIP158 >> => SCHEDCONST << EIP150 >>
      requires notBool      ( SCHEDCONST ==K Gemptyisnonexistent orBool SCHEDCONST␣
↪==K Gzerovaluenewaccountgas )
```

```
static const EVMSchedule EIP158Schedule = []
{
    EVMSchedule schedule = EIP150Schedule;
    schedule.expByteGas = 50;
    schedule.eip158Mode = true;
    schedule.maxCodeSize = 0x6000;
    return schedule;
}();
```

5.3.8 Byzantium Schedule

```
    syntax Schedule ::= "BYZANTIUM"
 // -------------------------------
    rule Rb         < BYZANTIUM > => 3 *Int (10 ^Int 18)
    rule SCHEDCONST < BYZANTIUM > => SCHEDCONST < EIP158 >
      requires notBool ( SCHEDCONST ==K Rb )

    rule Ghasrevert     << BYZANTIUM >> => true
    rule Ghasreturndata << BYZANTIUM >> => true
    rule Ghasstaticcall << BYZANTIUM >> => true
    rule SCHEDFLAG      << BYZANTIUM >> => SCHEDFLAG << EIP158 >>
      requires notBool ( SCHEDFLAG ==K Ghasrevert orBool SCHEDFLAG ==K Ghasreturndata␣
↪orBool SCHEDFLAG ==K Ghasstaticcall )
```

```
static const EVMSchedule ByzantiumSchedule = []
{
    EVMSchedule schedule = EIP158Schedule;
    schedule.haveRevert = true;
    schedule.haveReturnData = true;
    schedule.haveStaticCall = true;
    schedule.blockRewardOverwrite = {3 * ether};
    return schedule;
}();
```

5.3.9 Constantinople Schedule

```
    syntax Schedule ::= "CONSTANTINOPLE"
 // ------------------------------------
    rule Gblockhash < CONSTANTINOPLE > => 800
    rule SCHEDCONST < CONSTANTINOPLE > => SCHEDCONST < BYZANTIUM >
      requires SCHEDCONST =/=K Gblockhash

    rule SCHEDFLAG << CONSTANTINOPLE >> => SCHEDFLAG << BYZANTIUM >>
```

```
static const EVMSchedule ConstantinopleSchedule = []
{
    EVMSchedule schedule = ByzantiumSchedule;
    schedule.blockhashGas = 800;
    schedule.haveCreate2 = true;
    return schedule;
}();
```

CHAPTER

SIX

EVM PROGRAM REPRESENTATIONS

EVM programs are represented algebraically in K, but programs can load and manipulate program data directly. The opcodes `CODECOPY` and `EXTCODECOPY` rely on the assembled form of the programs being present. The opcode `CREATE` relies on being able to interperet EVM data as a program.

This is a program representation dependence, which we might want to avoid. Perhaps the only program representation dependence we should have is the hash of the program; doing so achieves:

- Program representation independence (different analysis tools on the language don't have to ensure they have a common representation of programs, just a common interperetation of the data-files holding programs).
- Programming language independence (we wouldn't even have to commit to a particular language or interperetation of the data-file).
- Only depending on the hash allows us to know that we have *exactly* the correct data-file (program), and nothing more.

6.1 Disassembler

After interpreting the strings representing programs as a `WordStack`, it should be changed into an `OpCodes` for use by the EVM semantics.

- `#dasmOpCodes` interperets `WordStack` as an `OpCodes`.
- `#dasmPUSH` handles the case of a `PushOp`.
- `#dasmOpCode` interperets a `Int` as an `OpCode`.

```
    syntax OpCodes ::= #dasmOpCodes ( WordStack , Schedule )            [function]
                     | #dasmOpCodes ( OpCodes , WordStack , Schedule ) [function,␣
↪klabel(#dasmOpCodesAux)]
                     | #revOpCodes  ( OpCodes , OpCodes )               [function]
 // ------------------------------------------------------------------------------
    rule #dasmOpCodes( WS, SCHED ) => #revOpCodes(#dasmOpCodes(.OpCodes, WS, SCHED), .
↪OpCodes)

    rule #dasmOpCodes( OPS, .WordStack, _ ) => OPS
    rule #dasmOpCodes( OPS, W : WS, SCHED ) => #dasmOpCodes(#dasmOpCode(W, SCHED) ;␣
↪OPS, WS, SCHED) requires W >=Int 0   andBool W <=Int 95
    rule #dasmOpCodes( OPS, W : WS, SCHED ) => #dasmOpCodes(#dasmOpCode(W, SCHED) ;␣
↪OPS, WS, SCHED) requires W >=Int 165 andBool W <=Int 255
    rule #dasmOpCodes( OPS, W : WS, SCHED ) => #dasmOpCodes(DUP(W -Int 127)       ;␣
↪OPS, WS, SCHED) requires W >=Int 128 andBool W <=Int 143
    rule #dasmOpCodes( OPS, W : WS, SCHED ) => #dasmOpCodes(SWAP(W -Int 143)      ;␣
↪OPS, WS, SCHED) requires W >=Int 144 andBool W <=Int 159
```

(continues on next page)

(continued from previous page)

```
    rule #dasmOpCodes( OPS, W : WS, SCHED ) => #dasmOpCodes(LOG(W -Int 160)       ;␣
↪OPS, WS, SCHED) requires W >=Int 160 andBool W <=Int 164

    rule #dasmOpCodes( OPS, W : WS, SCHED ) => #dasmOpCodes(PUSH(W -Int 95, #asWord(
↪#take(W -Int 95, WS))) ; OPS, #drop(W -Int 95, WS), SCHED) requires W >=Int 96 ␣
↪andBool W <=Int 127

    rule #revOpCodes ( OP ; OPS , OPS' ) => #revOpCodes(OPS, OP ; OPS')
    rule #revOpCodes ( .OpCodes , OPS  ) => OPS

    syntax OpCode ::= #dasmOpCode ( Int , Schedule ) [function]
 // -----------------------------------------------------------
    rule #dasmOpCode(   0,     _ ) => STOP
    rule #dasmOpCode(   1,     _ ) => ADD
    rule #dasmOpCode(   2,     _ ) => MUL
    rule #dasmOpCode(   3,     _ ) => SUB
    rule #dasmOpCode(   4,     _ ) => DIV
    rule #dasmOpCode(   5,     _ ) => SDIV
    rule #dasmOpCode(   6,     _ ) => MOD
    rule #dasmOpCode(   7,     _ ) => SMOD
    rule #dasmOpCode(   8,     _ ) => ADDMOD
    rule #dasmOpCode(   9,     _ ) => MULMOD
    rule #dasmOpCode(  10,     _ ) => EXP
    rule #dasmOpCode(  11,     _ ) => SIGNEXTEND
    rule #dasmOpCode(  16,     _ ) => LT
    rule #dasmOpCode(  17,     _ ) => GT
    rule #dasmOpCode(  18,     _ ) => SLT
    rule #dasmOpCode(  19,     _ ) => SGT
    rule #dasmOpCode(  20,     _ ) => EQ
    rule #dasmOpCode(  21,     _ ) => ISZERO
    rule #dasmOpCode(  22,     _ ) => AND
    rule #dasmOpCode(  23,     _ ) => EVMOR
    rule #dasmOpCode(  24,     _ ) => XOR
    rule #dasmOpCode(  25,     _ ) => NOT
    rule #dasmOpCode(  26,     _ ) => BYTE
    rule #dasmOpCode(  32,     _ ) => SHA3
    rule #dasmOpCode(  48,     _ ) => ADDRESS
    rule #dasmOpCode(  49,     _ ) => BALANCE
    rule #dasmOpCode(  50,     _ ) => ORIGIN
    rule #dasmOpCode(  51,     _ ) => CALLER
    rule #dasmOpCode(  52,     _ ) => CALLVALUE
    rule #dasmOpCode(  53,     _ ) => CALLDATALOAD
    rule #dasmOpCode(  54,     _ ) => CALLDATASIZE
    rule #dasmOpCode(  55,     _ ) => CALLDATACOPY
    rule #dasmOpCode(  56,     _ ) => CODESIZE
    rule #dasmOpCode(  57,     _ ) => CODECOPY
    rule #dasmOpCode(  58,     _ ) => GASPRICE
    rule #dasmOpCode(  59,     _ ) => EXTCODESIZE
    rule #dasmOpCode(  60,     _ ) => EXTCODECOPY
    rule #dasmOpCode(  61, SCHED ) => RETURNDATASIZE requires Ghasreturndata << SCHED␣
↪>>
    rule #dasmOpCode(  62, SCHED ) => RETURNDATACOPY requires Ghasreturndata << SCHED␣
↪>>
    rule #dasmOpCode(  64,     _ ) => BLOCKHASH
    rule #dasmOpCode(  65,     _ ) => COINBASE
    rule #dasmOpCode(  66,     _ ) => TIMESTAMP
    rule #dasmOpCode(  67,     _ ) => NUMBER
```

(continues on next page)

(continued from previous page)

```
    rule #dasmOpCode(  68,     _ ) => DIFFICULTY
    rule #dasmOpCode(  69,     _ ) => GASLIMIT
    rule #dasmOpCode(  80,     _ ) => POP
    rule #dasmOpCode(  81,     _ ) => MLOAD
    rule #dasmOpCode(  82,     _ ) => MSTORE
    rule #dasmOpCode(  83,     _ ) => MSTORE8
    rule #dasmOpCode(  84,     _ ) => SLOAD
    rule #dasmOpCode(  85,     _ ) => SSTORE
    rule #dasmOpCode(  86,     _ ) => JUMP
    rule #dasmOpCode(  87,     _ ) => JUMPI
    rule #dasmOpCode(  88,     _ ) => PC
    rule #dasmOpCode(  89,     _ ) => MSIZE
    rule #dasmOpCode(  90,     _ ) => GAS
    rule #dasmOpCode(  91,     _ ) => JUMPDEST
    rule #dasmOpCode( 240,     _ ) => CREATE
    rule #dasmOpCode( 241,     _ ) => CALL
    rule #dasmOpCode( 242,     _ ) => CALLCODE
    rule #dasmOpCode( 243,     _ ) => RETURN
    rule #dasmOpCode( 244, SCHED ) => DELEGATECALL requires SCHED =/=K FRONTIER
    rule #dasmOpCode( 250, SCHED ) => STATICCALL   requires Ghasstaticcall << SCHED >>
    rule #dasmOpCode( 253, SCHED ) => REVERT       requires Ghasrevert     << SCHED >>
    rule #dasmOpCode( 254,     _ ) => INVALID
    rule #dasmOpCode( 255,     _ ) => SELFDESTRUCT
    rule #dasmOpCode(   W,     _ ) => UNDEFINED(W)  [owise]
endmodule
```

CHAPTER
SEVEN

EVM INTEGRATION WITH PRODUCTION CLIENT

Contained in this file is glue code needed in order to enable the ability to use KEVM as a VM for an actual Ethereum node.

```
module EVM-NODE
    imports EVM
    imports K-REFLECTION
    imports COLLECTIONS
```

7.1 State loading operations.

In order to enable scalable execution of transactions on an entire blockchain, it is necessary to avoid serializing/deserializing the entire state of all accounts when constructing the initial configuration for KEVM. To do this, we assume that accounts not present in the `<accounts>` cell might not exist and need to be loaded on each access. We also defer loading of storage entries and the actual code byte string until it is needed. Because the same account may be loaded more than once, implementations of this interface are expected to cache the actual query to the Ethereum client.

- `#unloaded` represents the code of an account that has not had its code loaded yet. Unloaded code may not be empty.
- Empty code is detected without lazy evaluation by means of checking the code hash, and therefore will always be represented in the `<code>` cell as `.WordStack`.

```
    syntax AccountCode ::= "#unloaded"
```

- `#getBalance` returns the balance of an account that exists based on its integer address.
- `#getNonce` returns the nonce of an account that exists based on its integer address.
- `#isCodeEmpty` returns true if the code hash of the account is equal to the hash of the empty string, and false otherwise.
- `#accountExists` returns true if the account is present in the state trie for the current block, and false otherwise.

```
    syntax Int  ::= #getBalance ( Int ) [function, hook(BLOCKCHAIN.getBalance)]
                  | #getNonce   ( Int ) [function, hook(BLOCKCHAIN.getNonce)]
 // ------------------------------------------------------------------------

    syntax Bool ::= #isCodeEmpty   ( Int ) [function, hook(BLOCKCHAIN.isCodeEmpty)]
                  | #accountExists ( Int ) [function, hook(BLOCKCHAIN.accountExists)]
 // --------------------------------------------------------------------------------
```

- `#loadAccount` loads an account's balance and nonce if it exists, and leaves the code and storage unloaded, except if the code is empty, in which case the code is fully loaded. If the account does not exist, it does nothing.

```
    rule <k> #loadAccount ACCT => . ... </k>
         <activeAccounts> ACCTS (.Set => SetItem(ACCT)) </activeAccounts>
         <accounts>
           ( .Bag
          => <account>
               <acctID> ACCT </acctID>
               <balance> #getBalance(ACCT) </balance>
               <code> #if #isCodeEmpty(ACCT) #then .WordStack #else #unloaded #fi </
↪code>
               <storage> .Map </storage>
               <nonce> #getNonce(ACCT) </nonce>
             </account>
           )
           ...
         </accounts>
      requires notBool ACCT in ACCTS andBool #accountExists(ACCT)

    rule <k> #loadAccount ACCT => . ... </k>
         <activeAccounts> ACCTS </activeAccounts>
      requires ACCT in ACCTS orBool notBool #accountExists(ACCT)
```

- `#getStorageData` loads the value for a single storage key of a specified account by its address and storage offset. If the storage key has already been loaded or the account does not exist, it does nothing.

```
    syntax Int ::= #getStorageData ( Int , Int ) [function, hook(BLOCKCHAIN.
↪getStorageData)]
 // ---------------------------------------------------------------------------------
↪------
    rule <k> #lookupStorage ACCT INDEX => . ... </k>
         <account>
           <acctID>  ACCT                                                           </
↪acctID>
           <storage> STORAGE => STORAGE [ INDEX <- #getStorageData(ACCT, INDEX) ] </
↪storage>
           ...
         </account>
      requires notBool INDEX in_keys(STORAGE)

    rule <k> #lookupStorage ACCT INDEX => . ... </k>
         <account>
           <acctID> ACCT </acctID>
           <storage> ... INDEX |-> _ ... </storage>
           ...
         </account>

    rule <k> #lookupStorage ACCT _ => . ... </k>
      requires notBool #accountExists(ACCT)
```

- `#getCode` loads the code for a specified account by its address. If the code has already been loaded, it does nothing. If the account does not exist, it also does nothing.

```
    syntax String ::= #getCode ( Int ) [function, hook(BLOCKCHAIN.getCode)]
 // ------------------------------------------------------------------------
    rule <k> #lookupCode ACCT => . ... </k>
         <account>
```

(continues on next page)

(continued from previous page)

```
          <acctID> ACCT </acctID>
          <code> #unloaded => #parseByteStackRaw(#getCode(ACCT)) </code>
          ...
        </account>

    rule <k> #lookupCode ACCT => . ... </k>
        <account>
          <acctID> ACCT </acctID>
          <code> _:WordStack </code>
          ...
        </account>

    rule <k> #lookupCode ACCT => . ... </k>
      requires notBool #accountExists(ACCT)
```

- `#getBlockhash(N)` returns the blockhash of the Nth most recent block, up to a maximum of 256 blocks. It is used in the implementation of the BLOCKHASH instruction as seen below.

```
    syntax Int ::= #getBlockhash ( Int ) [function, hook(BLOCKCHAIN.getBlockhash)]
 // ------------------------------------------------------------------------------
    rule <k> BLOCKHASH N => #getBlockhash(N) ~> #push ... </k> <mode> NORMAL </mode>␣
↪requires N >=Int 0 andBool N  <Int 256
    rule <k> BLOCKHASH N => 0                ~> #push ... </k> <mode> NORMAL </mode>␣
↪requires N  <Int 0  orBool N >=Int 256
```

7.2 Transaction Execution

- `runVM` takes all the input state of a transaction and the current block header and executes the transaction according to the specified state, relying on the above loading operations for access to accounts and block hashes. The signature of this function must match the signature expected by VM.ml in the blockchain-k-plugin.

```
    syntax EthereumSimulation ::= runVM ( iscreate: Bool , to: Int          , from:␣
↪Int        , code: String  , args: String  , value: Int      , gasprice: Int
                                                      , gas: Int        , beneficiary: Int ,␣
↪difficulty: Int , number: Int   , gaslimit: Int , timestamp: Int , unused: String )
 // ------------------------------------------------------------------------------------
↪---------------------------------------------------------------------------
    rule <k> (.K => #loadAccount ACCTFROM) ~> runVM(... from: ACCTFROM) ... </k>
         <activeAccounts> .Set </activeAccounts>

    rule <k> runVM(true, _, ACCTFROM, _, ARGS, VALUE, GPRICE, GAVAIL, CB, DIFF, NUMB,␣
↪GLIMIT, TS, _)
          => #loadAccount #newAddr(ACCTFROM, NONCE -Int 1)
          ~> #create ACCTFROM #newAddr(ACCTFROM, NONCE -Int 1) GAVAIL VALUE
↪#parseByteStackRaw(ARGS)
          ~> #codeDeposit #newAddr(ACCTFROM, NONCE -Int 1)
          ~> #endCreate
         ...
         </k>
         <schedule> SCHED </schedule>
         <gasPrice> _ => GPRICE </gasPrice>
         <origin> _ => ACCTFROM </origin>
         <callDepth> _ => -1 </callDepth>
         <coinbase> _ => CB </coinbase>
```

(continues on next page)

(continued from previous page)

```
         <difficulty> _ => DIFF </difficulty>
         <number> _ => NUMB </number>
         <gasLimit> _ => GLIMIT </gasLimit>
         <timestamp> _ => TS </timestamp>
         <account>
           <acctID> ACCTFROM </acctID>
           <nonce> NONCE </nonce>
           ...
         </account>
         <touchedAccounts> _ => SetItem(CB) </touchedAccounts>
         <activeAccounts> ACCTS </activeAccounts>
      requires ACCTFROM in ACCTS

    rule <k> runVM(false, ACCTTO, ACCTFROM, _, ARGS, VALUE, GPRICE, GAVAIL, CB, DIFF,␣
↪NUMB, GLIMIT, TS, _)
          => #loadAccount ACCTTO
          ~> #lookupCode ACCTTO
          ~> #call ACCTFROM ACCTTO ACCTTO GAVAIL VALUE VALUE #parseByteStackRaw(ARGS)␣
↪false
          ~> #endVM
         ...
         </k>
         <schedule> SCHED </schedule>
         <gasPrice> _ => GPRICE </gasPrice>
         <origin> _ => ACCTFROM </origin>
         <callDepth> _ => -1 </callDepth>
         <coinbase> _ => CB </coinbase>
         <difficulty> _ => DIFF </difficulty>
         <number> _ => NUMB </number>
         <gasLimit> _ => GLIMIT </gasLimit>
         <timestamp> _ => TS </timestamp>
         <touchedAccounts> _ => SetItem(CB) </touchedAccounts>
         <activeAccounts> ACCTS </activeAccounts>
      requires ACCTFROM in ACCTS
```

- `#endCreate` and `#endVM` clean up after the transaction finishes and store the return status code of the top level call frame on the top of the `<k>` cell.

```
    syntax KItem ::= "#endVM" | "#endCreate"
 // ----------------------------------------
    rule <statusCode> _:ExceptionalStatusCode </statusCode>
         <k> #halt ~> #endVM => #popCallStack ~> #popWorldState ~> 0 </k>
         <output> _ => .WordStack </output>

    rule <statusCode> EVMC_REVERT </statusCode>
         <k> #halt ~> #endVM => #popCallStack ~> #popWorldState ~> #refund GAVAIL ~>␣
↪0 </k>
         <gas> GAVAIL </gas>

    rule <statusCode> EVMC_SUCCESS </statusCode>
         <k> #halt ~> #endVM => #popCallStack ~> #dropWorldState ~> #refund GAVAIL ~>␣
↪1 </k>
         <gas> GAVAIL </gas>

    rule <k> #endCreate => W ... </k> <wordStack> W : WS </wordStack>
```

7.3 Primitive operations expected to exist by the blockchain-k-plugin

- `vmResult` represents the extracted information about the world state after the transaction finishes. Its signature must match the signature expected by VM.ml in the blockchain-k-plugin.
- `extractConfig` takes a final configuration after rewriting and extracts a `vmResult` from it in order to abstract away configuration structure from the postprocessing done by the blockchain-k-plugin.

```
    syntax KItem ::= vmResult ( return: String , gas: Int , refund: Int , status: Int␣
↪, selfdestruct: List , logs: List , AccountsCell , touched: List )
    syntax KItem ::= extractConfig ( GeneratedTopCell ) [function]
 // -------------------------------------------------------------
    rule extractConfig ( <generatedTop>
                           <output> OUT </output>
                           <gas> GAVAIL </gas>
                           <refund> REFUND </refund>
                           <k> STATUS:Int </k>
                           <selfDestruct> SD </selfDestruct>
                           <log> LOGS </log>
                           <accounts> ACCTS </accounts>
                           <touchedAccounts> TOUCHED </touchedAccounts>
                           ...
                         </generatedTop>
                       )
      => vmResult(#unparseByteStack(OUT), GAVAIL, REFUND, STATUS, Set2List(SD), LOGS,
↪<accounts> ACCTS </accounts>, Set2List(TOUCHED))
```

- `contractBytes` takes the contents of the `<code>` cell and returns its binary representation as a String.

```
    syntax String ::= contractBytes(WordStack) [function]
 // ----------------------------------------------------
    rule contractBytes(WS) => #unparseByteStack(WS)
```

The following are expected to exist in the client, but are already defined in [data.md].

- `accountEmpty` takes the contents of the `<code>` cell, the contents of the `<nonce>` cell, and the contents of the `<balance>` cell and returns true if the account is empty according to the semantics of EIP161 (i.e., empty code zero balance zero nonce).
- `unparseByteStack` takes a WordStack and returns the corresponding byte String.
- `initGeneratedTopCell` is the top cell initializer used to construct an initial configuration. The configuration is expected to have `$MODE`, `$PGM`, and `$SCHEDULE` parameters.
- `logEntry` is an entry in the log data created by a transaction. It is expected to consist of an Int address, a List of Int topics, and a WordStack of data.
- `NORMAL` is the value of `$MODE` used by actual transaction execution.

```
endmodule
```

CHAPTER

EIGHT

ETHEREUM SIMULATIONS

Ethereum is using the EVM to drive updates over the world state. Actual execution of the EVM is defined in the EVM file.

```
requires "evm-node.k"
```

```
requires "evm.k"
requires "analysis.k"

module ETHEREUM-SIMULATION
    imports EVM
    imports K-REFLECTION
```

```
    imports EVM-ANALYSIS
```

```
    imports EVM-NODE
```

An Ethereum simulation is a list of Ethereum commands. Some Ethereum commands take an Ethereum specification (eg. for an account or transaction).

```
    syntax EthereumSimulation ::= ".EthereumSimulation"
                                | EthereumCommand EthereumSimulation
 // ------------------------------------------------------------------
    rule <k> .EthereumSimulation                              => .            ␣
↪   ... </k>
    rule <k> ETC                       ETS:EthereumSimulation => ETC        ~>␣
↪ETS  ... </k>
    rule <k> ETC1:EthereumCommand ~> ETC2 ETS:EthereumSimulation => ETC1 ~> ETC2 ~>␣
↪ETS  ... </k>
    rule <k> KI:KItem             ~> ETC2 ETS:EthereumSimulation => KI   ~> ETC2 ~>␣
↪ETS  ... </k>

    syntax EthereumSimulation ::= JSON
 // ----------------------------------
    rule <k> JSONINPUT:JSON => run JSONINPUT success .EthereumSimulation </k>
```

For verification purposes, it's much easier to specify a program in terms of its op-codes and not the hex-encoding that the tests use. To do so, we'll extend sort `JSON` with some EVM specific syntax, and provide a "pretti-fication" to the nicer input form.

```
    syntax JSON ::= Int | WordStack | OpCodes | Map | Call | SubstateLogEntry |␣
↪Account
 // ---------------------------------------------------------------------------
↪-
```

(continues on next page)

(continued from previous page)

```
    syntax JSONList ::= #sortJSONList ( JSONList )            [function]
                      | #sortJSONList ( JSONList , JSONList ) [function, klabel(
↪#sortJSONListAux)]
 // -----------------------------------------------------------------------------
↪------------
    rule #sortJSONList(JS) => #sortJSONList(JS, .JSONList)
    rule #sortJSONList(.JSONList, LS)            => LS
    rule #sortJSONList(((KEY : VAL) , REST), LS) => #insertJSONKey((KEY : VAL),
↪#sortJSONList(REST, LS))

    syntax JSONList ::= #insertJSONKey ( JSON , JSONList ) [function]
 // -----------------------------------------------------------------
    rule #insertJSONKey( JS , .JSONList ) => JS , .JSONList
    rule #insertJSONKey( (KEY : VAL) , ((KEY' : VAL') , REST) ) => (KEY : VAL)   ,␣
↪(KEY' : VAL')                 , REST  requires KEY <String KEY'
    rule #insertJSONKey( (KEY : VAL) , ((KEY' : VAL') , REST) ) => (KEY' : VAL') ,
↪#insertJSONKey((KEY : VAL) , REST) requires KEY >=String KEY'

    syntax Bool ::= #isSorted ( JSONList ) [function]
 // -------------------------------------------------
    rule #isSorted( .JSONList ) => true
    rule #isSorted( KEY : _ )   => true
    rule #isSorted( (KEY : _) , (KEY' : VAL) , REST ) => KEY <=String KEY'␣
↪andThenBool #isSorted((KEY' : VAL) , REST)
```

8.1 Driving Execution

- `start` places `#next` on the `<k>` cell so that execution of the loaded state begin.
- `flush` places `#finalize` on the `<k>` cell.

```
    syntax EthereumCommand ::= "start"
 // ----------------------------------
    rule <mode> NORMAL      </mode> <k> start => #execute    ... </k>
    rule <mode> VMTESTS     </mode> <k> start => #execute    ... </k>
    rule <mode> GASANALYZE </mode> <k> start => #gasAnalyze ... </k>

    syntax EthereumCommand ::= "flush"
 // ----------------------------------
    rule <mode> EXECMODE </mode> <statusCode> EVMC_SUCCESS            </statusCode>
↪<k> #halt ~> flush => #finalizeTx(EXECMODE ==K VMTESTS)          ... </k>
    rule <mode> EXECMODE </mode> <statusCode> _:ExceptionalStatusCode </statusCode>
↪<k> #halt ~> flush => #finalizeTx(EXECMODE ==K VMTESTS) ~> #halt ... </k>
```

- `startTx` computes the sender of the transaction, and places loadTx on the `k` cell.
- `loadTx(_)` loads the next transaction to be executed into the current state.
- `finishTx` is a place-holder for performing necessary cleanup after a transaction.

TODO: `loadTx(_) => loadTx_`

```
    syntax EthereumCommand ::= "startTx"
 // ------------------------------------
```

(continues on next page)

(continued from previous page)

```
    rule <k> startTx => #finalizeBlock ... </k>
         <txPending> .List </txPending>

    rule <k> startTx => loadTx(#sender(TN, TP, TG, TT, TV, #unparseByteStack(DATA),␣
↪TW, TR, TS)) ... </k>
         <txPending> ListItem(TXID:Int) ... </txPending>
         <message>
           <msgID>      TXID </msgID>
           <txNonce>    TN   </txNonce>
           <txGasPrice> TP   </txGasPrice>
           <txGasLimit> TG   </txGasLimit>
           <to>         TT   </to>
           <value>      TV   </value>
           <sigV>       TW   </sigV>
           <sigR>       TR   </sigR>
           <sigS>       TS   </sigS>
           <data>       DATA </data>
         </message>

    syntax EthereumCommand ::= loadTx ( Int )
 // -----------------------------------------
    rule <k> loadTx(ACCTFROM)
          => #loadAccount #newAddr(ACCTFROM, NONCE)
          ~> #create ACCTFROM #newAddr(ACCTFROM, NONCE) (GLIMIT -Int G0(SCHED, CODE,␣
↪true)) VALUE CODE
          ~> #execute ~> #finishTx ~> #finalizeTx(false) ~> startTx
         ...
         </k>
         <schedule> SCHED </schedule>
         <gasPrice> _ => GPRICE </gasPrice>
         <origin> _ => ACCTFROM </origin>
         <callDepth> _ => -1 </callDepth>
         <txPending> ListItem(TXID:Int) ... </txPending>
         <coinbase> MINER </coinbase>
         <message>
           <msgID>      TXID     </msgID>
           <txGasPrice> GPRICE   </txGasPrice>
           <txGasLimit> GLIMIT   </txGasLimit>
           <to>         .Account </to>
           <value>      VALUE    </value>
           <data>       CODE     </data>
           ...
         </message>
         <account>
           <acctID> ACCTFROM </acctID>
           <balance> BAL => BAL -Int (GLIMIT *Int GPRICE) </balance>
           <nonce> NONCE => NONCE +Int 1 </nonce>
           ...
         </account>
         <touchedAccounts> _ => SetItem(MINER) </touchedAccounts>

    rule <k> loadTx(ACCTFROM)
          => #loadAccount ACCTTO
          ~> #lookupCode  ACCTTO
          ~> #call ACCTFROM ACCTTO ACCTTO (GLIMIT -Int G0(SCHED, DATA, false)) VALUE␣
↪VALUE DATA false
          ~> #execute ~> #finishTx ~> #finalizeTx(false) ~> startTx
```

(continues on next page)

(continued from previous page)

```
         ...
         </k>
         <schedule> SCHED </schedule>
         <gasPrice> _ => GPRICE </gasPrice>
         <origin> _ => ACCTFROM </origin>
         <callDepth> _ => -1 </callDepth>
         <txPending> ListItem(TXID:Int) ... </txPending>
         <coinbase> MINER </coinbase>
         <message>
           <msgID>      TXID   </msgID>
           <txGasPrice> GPRICE </txGasPrice>
           <txGasLimit> GLIMIT </txGasLimit>
           <to>         ACCTTO </to>
           <value>      VALUE  </value>
           <data>       DATA   </data>
           ...
         </message>
         <account>
           <acctID> ACCTFROM </acctID>
           <balance> BAL => BAL -Int (GLIMIT *Int GPRICE) </balance>
           <nonce> NONCE => NONCE +Int 1 </nonce>
           ...
         </account>
         <touchedAccounts> _ => SetItem(MINER) </touchedAccounts>
      requires ACCTTO =/=K .Account

    syntax EthereumCommand ::= "#finishTx"
 // --------------------------------------
    rule <statusCode> _:ExceptionalStatusCode </statusCode> <k> #halt ~> #finishTx =>
↪#popCallStack ~> #popWorldState ... </k>
    rule <statusCode> EVMC_REVERT             </statusCode> <k> #halt ~> #finishTx =>
↪#popCallStack ~> #popWorldState ~> #refund GAVAIL ... </k> <gas> GAVAIL </gas>

    rule <statusCode> EVMC_SUCCESS </statusCode>
         <k> #halt ~> #finishTx => #mkCodeDeposit ACCT ... </k>
         <id> ACCT </id>
         <txPending> ListItem(TXID:Int) ... </txPending>
         <message>
           <msgID> TXID     </msgID>
           <to>    .Account </to>
           ...
         </message>

    rule <statusCode> EVMC_SUCCESS </statusCode>
         <k> #halt ~> #finishTx => #popCallStack ~> #dropWorldState ~> #refund GAVAIL␣
↪... </k>
         <id> ACCT </id>
         <gas> GAVAIL </gas>
         <txPending> ListItem(TXID:Int) ... </txPending>
         <message>
           <msgID> TXID </msgID>
           <to>    TT   </to>
           ...
         </message>
      requires TT =/=K .Account
```

- `#finalizeBlock` is used to signal that block finalization procedures should take place (after transactions have executed).

- `#rewardOmmers(_)` pays out the reward to uncle blocks so that blocks are orphaned less often in Ethereum.

```
    syntax EthereumCommand ::= "#finalizeBlock" | #rewardOmmers ( JSONList )
 // ------------------------------------------------------------------------
    rule <k> #finalizeBlock => #rewardOmmers(OMMERS) ... </k>
         <schedule> SCHED </schedule>
         <ommerBlockHeaders> [ OMMERS ] </ommerBlockHeaders>
         <coinbase> MINER </coinbase>
         <account>
           <acctID> MINER </acctID>
           <balance> MINBAL => MINBAL +Int Rb < SCHED > </balance>
           ...
         </account>

    rule <k> (.K => #newAccount MINER) ~> #finalizeBlock ... </k>
         <coinbase> MINER </coinbase>
         <activeAccounts> ACCTS </activeAccounts>
      requires notBool MINER in ACCTS

    rule <k> #rewardOmmers(.JSONList) => . ... </k>
    rule <k> #rewardOmmers([ _ , _ , OMMER , _ , _ , _ , _ , _ , OMMNUM , _ ] , REST)␣
↪=> #rewardOmmers(REST) ... </k>
         <schedule> SCHED </schedule>
         <coinbase> MINER </coinbase>
         <number> CURNUM </number>
         <account>
           <acctID> MINER </acctID>
           <balance> MINBAL => MINBAL +Int Rb < SCHED > /Int 32 </balance>
          ...
         </account>
         <account>
           <acctID> OMMER </acctID>
           <balance> OMMBAL => OMMBAL +Int Rb < SCHED > +Int (OMMNUM -Int CURNUM)␣
↪*Int (Rb < SCHED > /Int 8) </balance>
          ...
         </account>
```

- `exception` only clears from the `<k>` cell if there is an exception preceding it.
- `failure_` holds the name of a test that failed if a test does fail.
- `success` sets the `<exit-code>` to `0` and the `<mode>` to `SUCCESS`.

```
    syntax Mode ::= "SUCCESS"
 // -------------------------

    syntax EthereumCommand ::= "exception" | "status" StatusCode
 // ------------------------------------------------------------
    rule <statusCode> _:ExceptionalStatusCode </statusCode>
         <k> #halt ~> exception => . ... </k>

    rule <k> status SC => . ... </k> <statusCode> SC </statusCode>

    syntax EthereumCommand ::= "failure" String | "success"
 // -------------------------------------------------------
    rule <k> success => . ... </k> <exit-code> _ => 0 </exit-code> <mode> _ =>␣
↪SUCCESS </mode>
```

(continues on next page)

(continued from previous page)

```
    rule <k> failure _ => . ... </k>
```

8.2 Running Tests

- run runs a given set of Ethereum tests (from the test-set).

Note that TEST is sorted here so that key "network" comes before key "pre".

```
    syntax EthereumCommand ::= "run" JSON
 // --------------------------------------
    rule <k> run { .JSONList } => . ... </k>
    rule <k> run { TESTID : { TEST:JSONList } , TESTS }
          => run ( TESTID : { #sortJSONList(TEST) } )
          ~> #if #hasPost?( { TEST } ) #then .K #else exception #fi
          ~> clear
          ~> run { TESTS }
         ...
         </k>

    syntax Bool ::= "#hasPost?" "(" JSON ")" [function]
 // -----------------------------------------------------
    rule #hasPost? ({ .JSONList }) => false
    rule #hasPost? ({ (KEY:String) : _ , REST }) => (KEY in #postKeys) orBool
↪#hasPost? ({ REST })
```

- #loadKeys are all the JSON nodes which should be considered as loads before execution.

```
    syntax Set ::= "#loadKeys" [function]
 // --------------------------------------
    rule #loadKeys => ( SetItem("env") SetItem("pre") SetItem("rlp") SetItem("network
↪") SetItem("genesisRLP") )

    rule <k> run TESTID : { KEY : (VAL:JSON) , REST } => load KEY : VAL ~> run TESTID␣
↪: { REST } ... </k>
      requires KEY in #loadKeys

    rule <k> run TESTID : { "blocks" : [ { KEY : VAL , REST1 => REST1 }, .JSONList ] ,
↪ ( REST2 => KEY : VAL , REST2 ) } ... </k>
    rule <k> run TESTID : { "blocks" : [ { .JSONList }, .JSONList ] , REST } => run␣
↪TESTID : { REST }                            ... </k>
```

- #execKeys are all the JSON nodes which should be considered for execution (between loading and checking).

```
    syntax Set ::= "#execKeys" [function]
 // --------------------------------------
    rule #execKeys => ( SetItem("exec") SetItem("lastblockhash") )

    rule <k> run TESTID : { KEY : (VAL:JSON) , NEXT , REST } => run TESTID : { NEXT ,␣
↪KEY : VAL , REST } ... </k>
      requires KEY in #execKeys

    rule <k> run TESTID : { "exec" : (EXEC:JSON) } => load "exec" : EXEC ~> start ~>␣
↪flush ... </k>
    rule <k> run TESTID : { "lastblockhash" : (HASH:String) } => startTx              ␣
↪     ... </k>
```

- `#postKeys` are a subset of `#checkKeys` which correspond to post-state account checks.
- `#checkKeys` are all the JSON nodes which should be considered as checks after execution.

```
    syntax Set ::= "#postKeys" [function] | "#allPostKeys" [function] | "#checkKeys"␣
↪[function]
 // ------------------------------------------------------------------------------
↪---------
    rule #postKeys    => ( SetItem("post") SetItem("postState") )
    rule #allPostKeys => ( #postKeys SetItem("expect") SetItem("export") SetItem(
↪"expet") )
    rule #checkKeys   => ( #allPostKeys SetItem("logs") SetItem("out") SetItem("gas")
                           SetItem("blockHeader") SetItem("transactions") SetItem(
↪"uncleHeaders") SetItem("genesisBlockHeader")
                         )

    rule <k> run TESTID : { KEY : (VAL:JSON) , REST } => run TESTID : { REST } ~>␣
↪check TESTID : { "post" : VAL } ... </k> requires KEY in #allPostKeys
    rule <k> run TESTID : { KEY : (VAL:JSON) , REST } => run TESTID : { REST } ~>␣
↪check TESTID : { KEY    : VAL } ... </k> requires KEY in #checkKeys andBool notBool␣
↪KEY in #allPostKeys
```

- `#discardKeys` are all the JSON nodes in the tests which should just be ignored.

```
    syntax Set ::= "#discardKeys" [function]
 // ----------------------------------------
    rule #discardKeys => ( SetItem("//") SetItem("_info") SetItem("callcreates") )

    rule <k> run TESTID : { KEY : _ , REST } => run TESTID : { REST } ... </k>␣
↪requires KEY in #discardKeys
```

8.3 State Manipulation

8.3.1 Clearing State

- `clear` clears all the execution state of the machine.
- `clearX` clears the substate `X`, for `TX`, `BLOCK`, and `NETWORK`.

```
    syntax EthereumCommand ::= "clear"
 // ----------------------------------
    rule <k> clear => clearTX ~> clearBLOCK ~> clearNETWORK ... </k>
         <analysis> _ => .Map </analysis>

    syntax EthreumCommand ::= "clearTX"
 // -----------------------------------
    rule <k> clearTX => . ... </k>
         <output>            _ => .WordStack </output>
         <memoryUsed>        _ => 0          </memoryUsed>
         <callDepth>         _ => 0          </callDepth>
         <callStack>         _ => .List      </callStack>
         <program>           _ => .Map       </program>
         <programBytes>      _ => .WordStack </programBytes>
         <id>                _ => 0          </id>
         <caller>            _ => 0          </caller>
         <callData>          _ => .WordStack </callData>
```

(continues on next page)

(continued from previous page)

```
         <callValue>       _ => 0          </callValue>
         <wordStack>       _ => .WordStack </wordStack>
         <localMem>        _ => .Map       </localMem>
         <pc>              _ => 0          </pc>
         <gas>             _ => 0          </gas>
         <previousGas>     _ => 0          </previousGas>
         <selfDestruct>    _ => .Set       </selfDestruct>
         <log>             _ => .List      </log>
         <refund>          _ => 0          </refund>
         <gasPrice>        _ => 0          </gasPrice>
         <origin>          _ => 0          </origin>
         <touchedAccounts> _ => .Set       </touchedAccounts>

    syntax EthreumCommand ::= "clearBLOCK"
 // --------------------------------------
    rule <k> clearBLOCK => . ... </k>
         <previousHash>      _ => 0              </previousHash>
         <ommersHash>        _ => 0              </ommersHash>
         <coinbase>          _ => 0              </coinbase>
         <stateRoot>         _ => 0              </stateRoot>
         <transactionsRoot>  _ => 0              </transactionsRoot>
         <receiptsRoot>      _ => 0              </receiptsRoot>
         <logsBloom>         _ => .WordStack     </logsBloom>
         <difficulty>        _ => 0              </difficulty>
         <number>            _ => 0              </number>
         <gasLimit>          _ => 0              </gasLimit>
         <gasUsed>           _ => 0              </gasUsed>
         <timestamp>         _ => 0              </timestamp>
         <extraData>         _ => .WordStack     </extraData>
         <mixHash>           _ => 0              </mixHash>
         <blockNonce>        _ => 0              </blockNonce>
         <ommerBlockHeaders> _ => [ .JSONList ] </ommerBlockHeaders>
         <blockhash>         _ => .List          </blockhash>

    syntax EthreumCommand ::= "clearNETWORK"
 // ----------------------------------------
    rule <k> clearNETWORK => . ... </k>
         <statusCode>     _ => .StatusCode </statusCode>
         <activeAccounts> _ => .Set        </activeAccounts>
         <accounts>       _ => .Bag        </accounts>
         <messages>       _ => .Bag        </messages>
         <schedule>       _ => DEFAULT     </schedule>
```

8.3.2 Loading State

- `mkAcct_` creates an account with the supplied ID (assuming it's already been chopped to 160 bits).

```
    syntax EthereumCommand ::= "mkAcct" Int
 // ---------------------------------------
    rule <k> mkAcct ACCT => #newAccount ACCT ... </k>
```

- `load` loads an account or transaction into the world state.

```
    syntax EthereumCommand ::= "load" JSON
 // --------------------------------------
```

(continues on next page)

(continued from previous page)

```
    rule <k> load DATA : { .JSONList }          => .                                    ␣
↪                 ... </k>
    rule <k> load DATA : { KEY : VALUE , REST } => load DATA : { KEY : VALUE } ~> ␣
↪load DATA : { REST } ... </k>
      requires REST =/=K .JSONList andBool DATA =/=String "transaction"

    rule <k> load DATA : [ .JSONList ]        => .                                      ␣
↪        ... </k>
    rule <k> load DATA : [ { TEST } , REST ] => load DATA : { TEST } ~> load DATA : [␣
↪REST ] ... </k>
```

Here we perform pre-proccesing on account data which allows “pretty” specification of input.

```
    rule <k> load "pre" : { (ACCTID:String) : ACCT }                => mkAcct
↪#parseAddr(ACCTID) ~> load "account" : { ACCTID : ACCT }                          ...
↪ </k>
    rule <k> load "account" : { ACCTID: { KEY : VALUE , REST } } => load "account" :
↪{ ACCTID : { KEY : VALUE } } ~> load "account" : { ACCTID : { REST } } ... </k>␣
↪requires REST =/=K .JSONList

    rule <k> load "account" : { ((ACCTID:String) => #parseAddr(ACCTID)) : ACCT }       ␣
↪                             ... </k>
    rule <k> load "account" : { (ACCT:Int) : { "balance" : ((VAL:String)         =>
↪#parseWord(VAL)) } }          ... </k>
    rule <k> load "account" : { (ACCT:Int) : { "nonce"   : ((VAL:String)         =>
↪#parseWord(VAL)) } }          ... </k>
    rule <k> load "account" : { (ACCT:Int) : { "code"    : ((CODE:String)        =>
↪#parseByteStack(CODE)) } }  ... </k>
    rule <k> load "account" : { (ACCT:Int) : { "storage" : ({ STORAGE:JSONList } =>
↪#parseMap({ STORAGE })) } } ... </k>
```

The individual fields of the accounts are dealt with here.

```
    rule <k> load "account" : { ACCT : { "balance" : (BAL:Int) } } => . ... </k>
         <account>
           <acctID> ACCT </acctID>
           <balance> _ => BAL </balance>
           ...
         </account>

    rule <k> load "account" : { ACCT : { "code" : (CODE:WordStack) } } => . ... </k>
         <account>
           <acctID> ACCT </acctID>
           <code> _ => CODE </code>
           ...
         </account>

    rule <k> load "account" : { ACCT : { "nonce" : (NONCE:Int) } } => . ... </k>
         <account>
           <acctID> ACCT </acctID>
           <nonce> _ => NONCE </nonce>
           ...
         </account>

    rule <k> load "account" : { ACCT : { "storage" : (STORAGE:Map) } } => . ... </k>
         <account>
           <acctID> ACCT </acctID>
```

(continues on next page)

(continued from previous page)

```
            <storage> _ => STORAGE </storage>
            ...
          </account>
```

Here we load the environmental information.

```
    rule <k> load "env" : { KEY : ((VAL:String) => #parseWord(VAL)) } ... </k>
      requires KEY in (SetItem("currentTimestamp") SetItem("currentGasLimit") SetItem(
↪"currentNumber") SetItem("currentDifficulty"))
    rule <k> load "env" : { KEY : ((VAL:String) => #parseHexWord(VAL)) } ... </k>
      requires KEY in (SetItem("currentCoinbase") SetItem("previousHash"))
 // ----------------------------------------------------------------------
    rule <k> load "env" : { "currentCoinbase"   : (CB:Int)     } => . ... </k>
↪<coinbase>      _ => CB      </coinbase>
    rule <k> load "env" : { "currentDifficulty" : (DIFF:Int)   } => . ... </k>
↪<difficulty>    _ => DIFF    </difficulty>
    rule <k> load "env" : { "currentGasLimit"   : (GLIMIT:Int) } => . ... </k>
↪<gasLimit>      _ => GLIMIT </gasLimit>
    rule <k> load "env" : { "currentNumber"     : (NUM:Int)    } => . ... </k>
↪<number>        _ => NUM     </number>
    rule <k> load "env" : { "previousHash"      : (HASH:Int)   } => . ... </k>
↪<previousHash> _ => HASH    </previousHash>
    rule <k> load "env" : { "currentTimestamp"  : (TS:Int)     } => . ... </k>
↪<timestamp>     _ => TS      </timestamp>

    rule <k> load "exec" : { KEY : ((VAL:String) => #parseWord(VAL)) } ... </k>
      requires KEY in (SetItem("gas") SetItem("gasPrice") SetItem("value"))
    rule <k> load "exec" : { KEY : ((VAL:String) => #parseHexWord(VAL)) } ... </k>
      requires KEY in (SetItem("address") SetItem("caller") SetItem("origin"))
 // --------------------------------------------------------------------------
    rule <k> load "exec" : { "gasPrice" : (GPRICE:Int)   } => . ... </k> <gasPrice>   _
↪ => GPRICE   </gasPrice>
    rule <k> load "exec" : { "gas"      : (GAVAIL:Int)   } => . ... </k> <gas>        _
↪ => GAVAIL   </gas>
    rule <k> load "exec" : { "address"  : (ACCTTO:Int)   } => . ... </k> <id>         _
↪ => ACCTTO   </id>
    rule <k> load "exec" : { "caller"   : (ACCTFROM:Int) } => . ... </k> <caller>     _
↪ => ACCTFROM </caller>
    rule <k> load "exec" : { "gas"      : (GAVAIL:Int)   } => . ... </k> <gas>        _
↪ => GAVAIL   </gas>
    rule <k> load "exec" : { "value"    : (VALUE:Int)    } => . ... </k> <callValue> _
↪ => VALUE    </callValue>
    rule <k> load "exec" : { "origin"   : (ORIG:Int)     } => . ... </k> <origin>     _
↪ => ORIG     </origin>
    rule <k> load "exec" : { "code"     : ((CODE:String)   => #parseByteStack(CODE)) }
↪ ... </k>

    rule <k> load "exec" : { "data" : ((DATA:String) => #parseByteStack(DATA)) } ...
↪</k>
 // ----------------------------------------------------------------------------------
↪---
    rule <k> load "exec" : { "data" : (DATA:WordStack) } => . ... </k> <callData> _ =>
↪ DATA </callData>
    rule <k> load "exec" : { "code" : (CODE:OpCodes)   } => . ... </k> <program>  _ =>
↪ #asMapOpCodes(CODE) </program>
    rule <k> load "exec" : { "code" : (CODE:WordStack) } => . ... </k> <program>  _ =>
↪ #asMapOpCodes(#dasmOpCodes(CODE, SCHED)) </program> <programBytes> _ => CODE </
↪programBytes> <schedule> SCHED </schedule>
```

(continues on next page)

(continued from previous page)

```
```

The "network" key allows setting the fee schedule inside the test.

```
    rule <k> load "network" : SCHEDSTRING => . ... </k>
         <schedule> _ => #asScheduleString(SCHEDSTRING) </schedule>

    syntax Schedule ::= #asScheduleString ( String ) [function]
 // -----------------------------------------------------------
    rule #asScheduleString("EIP150")         => EIP150
    rule #asScheduleString("EIP158")         => EIP158
    rule #asScheduleString("Frontier")       => FRONTIER
    rule #asScheduleString("Homestead")      => HOMESTEAD
    rule #asScheduleString("Byzantium")      => BYZANTIUM
    rule #asScheduleString("Constantinople") => CONSTANTINOPLE
```

The "rlp" key loads the block information.

```
    rule <k> load "rlp"        : (VAL:String => #rlpDecode(#unparseByteStack(
↪#parseByteStack(VAL)))) ... </k>
    rule <k> load "genesisRLP" : (VAL:String => #rlpDecode(#unparseByteStack(
↪#parseByteStack(VAL)))) ... </k>
 // ---------------------------------------------------------------------------------
↪-----------------------
    rule <k> load "rlp" : [ [ HP , HO , HC , HR , HT , HE , HB , HD , HI , HL , HG ,␣
↪HS , HX , HM , HN , .JSONList ] , BT , BU , .JSONList ]
          => load "transaction" : BT
         ...
         </k>
         <previousHash>      _ => #asWord(#parseByteStackRaw(HP)) </previousHash>
         <ommersHash>        _ => #asWord(#parseByteStackRaw(HO)) </ommersHash>
         <coinbase>          _ => #asWord(#parseByteStackRaw(HC)) </coinbase>
         <stateRoot>         _ => #asWord(#parseByteStackRaw(HR)) </stateRoot>
         <transactionsRoot>  _ => #asWord(#parseByteStackRaw(HT)) </transactionsRoot>
         <receiptsRoot>      _ => #asWord(#parseByteStackRaw(HE)) </receiptsRoot>
         <logsBloom>         _ => #parseByteStackRaw(HB)          </logsBloom>
         <difficulty>        _ => #asWord(#parseByteStackRaw(HD)) </difficulty>
         <number>            _ => #asWord(#parseByteStackRaw(HI)) </number>
         <gasLimit>          _ => #asWord(#parseByteStackRaw(HL)) </gasLimit>
         <gasUsed>           _ => #asWord(#parseByteStackRaw(HG)) </gasUsed>
         <timestamp>         _ => #asWord(#parseByteStackRaw(HS)) </timestamp>
         <extraData>         _ => #parseByteStackRaw(HX)          </extraData>
         <mixHash>           _ => #asWord(#parseByteStackRaw(HM)) </mixHash>
         <blockNonce>        _ => #asWord(#parseByteStackRaw(HN)) </blockNonce>
         <ommerBlockHeaders> _ => BU                              </ommerBlockHeaders>

    rule <k> load "genesisRLP": [ [ HP, HO, HC, HR, HT, HE:String, HB, HD, HI, HL, HG,
↪ HS, HX, HM, HN, .JSONList ], _, _, .JSONList ] => .K ... </k>
         <blockhash> .List => ListItem(#blockHeaderHash(HP, HO, HC, HR, HT, HE, HB,␣
↪HD, HI, HL, HG, HS, HX, HM, HN)) ListItem(#asWord(#parseByteStackRaw(HP))) ... </
↪blockhash>

    syntax EthereumCommand ::= "mkTX" Int
 // -------------------------------------
    rule <k> mkTX TXID => . ... </k>
         <txOrder>   ... (.List => ListItem(TXID)) </txOrder>
         <txPending> ... (.List => ListItem(TXID)) </txPending>
```

(continues on next page)

(continued from previous page)

```
          <messages>
            ( .Bag
           => <message>
                <msgID> TXID:Int </msgID>
                ...
              </message>
            )
           ...
           </messages>

     rule <k> load "transaction" : [ [ TN , TP , TG , TT , TV , TI , TW , TR , TS ] ,␣
↪REST ]
           => mkTX !ID:Int
           ~> load "transaction" : { !ID : { "data"  : TI   ,   "gasLimit" : TG   ,
↪"gasPrice" : TP
                                             , "nonce" : TN   ,   "r"        : TR   ,
↪"s"        : TS
                                             , "to"    : TT   ,   "v"        : TW   ,
↪"value"    : TV
                                             , .JSONList
                                             }
                                   }
           ~> load "transaction" : [ REST ]
           ...
           </k>

     rule <k> load "transaction" : { ACCTID: { KEY : VALUE , REST } }
           => load "transaction" : { ACCTID : { KEY : VALUE } }
           ~> load "transaction" : { ACCTID : { REST } }
           ...
          </k>
       requires REST =/=K .JSONList

     rule <k> load "transaction" : { TXID : { "gasLimit" : (TG:String => #asWord(
↪#parseByteStackRaw(TG)))          } } ... </k>
     rule <k> load "transaction" : { TXID : { "gasPrice" : (TP:String => #asWord(
↪#parseByteStackRaw(TP)))          } } ... </k>
     rule <k> load "transaction" : { TXID : { "nonce"    : (TN:String => #asWord(
↪#parseByteStackRaw(TN)))          } } ... </k>
     rule <k> load "transaction" : { TXID : { "v"        : (TW:String => #asWord(
↪#parseByteStackRaw(TW)))          } } ... </k>
     rule <k> load "transaction" : { TXID : { "value"    : (TV:String => #asWord(
↪#parseByteStackRaw(TV)))          } } ... </k>
     rule <k> load "transaction" : { TXID : { "to"       : (TT:String => #asAccount(
↪#parseByteStackRaw(TT)))       } } ... </k>
     rule <k> load "transaction" : { TXID : { "data"     : (TI:String =>
↪#parseByteStackRaw(TI))                  } } ... </k>
     rule <k> load "transaction" : { TXID : { "r"        : (TR:String =>
↪#padToWidth(32, #parseByteStackRaw(TR))) } } ... </k>
     rule <k> load "transaction" : { TXID : { "s"        : (TS:String =>
↪#padToWidth(32, #parseByteStackRaw(TS))) } } ... </k>

     rule <k> load "transaction" : { TXID : { "gasLimit" : TG:Int } } => . ... </k>
          <message> <msgID> TXID </msgID> <txGasLimit> _ => TG </txGasLimit> ... </
↪message>

     rule <k> load "transaction" : { TXID : { "gasPrice" : TP:Int } } => . ... </k>
```

(continues on next page)

(continued from previous page)

```
         <message> <msgID> TXID </msgID> <txGasPrice> _ => TP </txGasPrice> ... </
↪message>

    rule <k> load "transaction" : { TXID : { "nonce" : TN:Int } } => . ... </k>
         <message> <msgID> TXID </msgID> <txNonce> _ => TN </txNonce> ... </message>

    rule <k> load "transaction" : { TXID : { "value" : TV:Int } } => . ... </k>
         <message> <msgID> TXID </msgID> <value> _ => TV </value> ... </message>

    rule <k> load "transaction" : { TXID : { "to" : TT:Account } } => . ... </k>
         <message> <msgID> TXID </msgID> <to> _ => TT </to> ... </message>

    rule <k> load "transaction" : { TXID : { "data" : TI:WordStack } } => . ... </k>
         <message> <msgID> TXID </msgID> <data> _ => TI </data> ... </message>

    rule <k> load "transaction" : { TXID : { "v" : TW:Int } } => . ... </k>
         <message> <msgID> TXID </msgID> <sigV> _ => TW </sigV> ... </message>

    rule <k> load "transaction" : { TXID : { "r" : TR:WordStack } } => . ... </k>
         <message> <msgID> TXID </msgID> <sigR> _ => TR </sigR> ... </message>

    rule <k> load "transaction" : { TXID : { "s" : TS:WordStack } } => . ... </k>
         <message> <msgID> TXID </msgID> <sigS> _ => TS </sigS> ... </message>
```

8.3.3 Checking State

- `check_` checks if an account/transaction appears in the world-state as stated.

```
    syntax EthereumCommand ::= "check" JSON
 // ---------------------------------------
    rule <k> #halt ~> check J:JSON => check J ~> #halt ... </k>

    rule <k> check DATA : { .JSONList } => . ... </k> requires DATA =/=String
↪"transactions"
    rule <k> check DATA : [ .JSONList ] => . ... </k> requires DATA =/=String
↪"ommerHeaders"

    rule <k> check DATA : { (KEY:String) : VALUE , REST } => check DATA : { KEY :␣
↪VALUE } ~> check DATA : { REST } ... </k>
      requires REST =/=K .JSONList andBool notBool DATA in (SetItem("callcreates")␣
↪SetItem("transactions"))

    rule <k> check DATA : [ { TEST } , REST ] => check DATA : { TEST } ~> check DATA␣
↪: [ REST ] ... </k>
      requires DATA =/=String "transactions"

    rule <k> check (KEY:String) : { JS:JSONList => #sortJSONList(JS) } ... </k>
      requires KEY in (SetItem("callcreates")) andBool notBool #isSorted(JS)

    rule <k> check TESTID : { "post" : POST } => check "account" : POST ~> failure␣
↪TESTID ... </k>
    rule <k> check "account" : { ACCTID: { KEY : VALUE , REST } } => check "account"␣
↪: { ACCTID : { KEY : VALUE } } ~> check "account" : { ACCTID : { REST } } ... </k>
      requires REST =/=K .JSONList
```

(continues on next page)

(continued from previous page)

```
    rule <k> check "account" : { ((ACCTID:String) => #parseAddr(ACCTID)) : ACCT }    ␣
↪                           ... </k>
    rule <k> check "account" : { (ACCT:Int) : { "balance" : ((VAL:String)          =>
↪#parseWord(VAL)) } }          ... </k>
    rule <k> check "account" : { (ACCT:Int) : { "nonce"   : ((VAL:String)          =>
↪#parseWord(VAL)) } }          ... </k>
    rule <k> check "account" : { (ACCT:Int) : { "code"    : ((CODE:String)         =>
↪#parseByteStack(CODE)) } }   ... </k>
    rule <k> check "account" : { (ACCT:Int) : { "storage" : ({ STORAGE:JSONList } =>
↪#parseMap({ STORAGE })) } } ... </k>

    rule <mode> EXECMODE </mode>
         <k> check "account" : { ACCT : { "balance" : (BAL:Int) } } => . ... </k>
         <account>
           <acctID> ACCT </acctID>
           <balance> BAL </balance>
           ...
         </account>
      requires EXECMODE =/=K VMTESTS

    rule <mode> VMTESTS </mode>
         <k> check "account" : { ACCT : { "balance" : (BAL:Int) } } => . ... </k>

    rule <k> check "account" : { ACCT : { "nonce" : (NONCE:Int) } } => . ... </k>
         <account>
           <acctID> ACCT </acctID>
           <nonce> NONCE </nonce>
           ...
         </account>

    rule <k> check "account" : { ACCT : { "storage" : (STORAGE:Map) } } => . ... </k>
         <account>
           <acctID> ACCT </acctID>
           <storage> ACCTSTORAGE </storage>
           ...
         </account>
      requires #removeZeros(ACCTSTORAGE) ==K STORAGE

    rule <k> check "account" : { ACCT : { "code" : (CODE:WordStack) } } => . ... </k>
         <account>
           <acctID> ACCT </acctID>
           <code> CODE </code>
           ...
         </account>
```

Here we check the other post-conditions associated with an EVM test.

```
    rule <k> check TESTID : { "out" : OUT } => check "out" : OUT ~> failure TESTID ...
↪ </k>
 // ---------------------------------------------------------------------------------
↪-----
    rule <k> check "out" : ((OUT:String) => #parseByteStack(OUT)) ... </k>
    rule <k> check "out" : OUT => . ... </k> <output> OUT </output>

    rule <k> check TESTID : { "logs" : LOGS } => check "logs" : LOGS ~> failure␣
↪TESTID ... </k>
 // ---------------------------------------------------------------------------------
↪---------
```

(continues on next page)

(continued from previous page)

```
    rule <k> check "logs" : HASH:String => . ... </k> <log> SL </log> requires
↪#parseHexBytes(Keccak256(#rlpEncodeLogs(SL))) ==K #parseByteStack(HASH)

    syntax String ::= #rlpEncodeLogs(List)        [function]
                    | #rlpEncodeLogsAux(List)     [function]
                    | #rlpEncodeTopics(List)      [function]
 // -----------------------------------------------------
    rule #rlpEncodeLogs(SL) => #rlpEncodeLength(#rlpEncodeLogsAux(SL), 192)
    rule #rlpEncodeLogsAux(ListItem({ ACCT | TOPICS | DATA }) SL) => #rlpEncodeLength(
↪#rlpEncodeBytes(ACCT, 20) +String #rlpEncodeLength(#rlpEncodeTopics(TOPICS), 192)␣
↪+String #rlpEncodeString(#unparseByteStack(DATA)), 192) +String
↪#rlpEncodeLogsAux(SL)
    rule #rlpEncodeLogsAux(.List) => ""
    rule #rlpEncodeTopics(ListItem(TOPIC) TOPICS) => #rlpEncodeBytes(TOPIC, 32)␣
↪+String #rlpEncodeTopics(TOPICS)
    rule #rlpEncodeTopics(.List) => ""

    rule <k> check TESTID : { "gas" : GLEFT } => check "gas" : GLEFT ~> failure␣
↪TESTID ... </k>
 // ------------------------------------------------------------------------
↪---------
    rule <k> check "gas" : ((GLEFT:String) => #parseWord(GLEFT)) ... </k>
    rule <k> check "gas" : GLEFT => . ... </k> <gas> GLEFT </gas>

    rule check TESTID : { "blockHeader" : BLOCKHEADER } => check "blockHeader" :␣
↪BLOCKHEADER ~> failure TESTID
 // ------------------------------------------------------------------------
↪------------------------
    rule <k> check "blockHeader" : { KEY : VALUE , REST } => check "blockHeader" : {␣
↪KEY : VALUE } ~> check "blockHeader" : { REST } ... </k>
      requires REST =/=K .JSONList

    rule <k> check "blockHeader" : { KEY : (VALUE:String => #parseByteStack(VALUE)) }␣
↪... </k>

    rule <k> check "blockHeader" : { KEY : (VALUE:WordStack => #asWord(VALUE)) } ...
↪</k>
      requires KEY in ( SetItem("coinbase") SetItem("difficulty") SetItem("gasLimit")␣
↪SetItem("gasUsed")
                        SetItem("mixHash") SetItem("nonce") SetItem("number") SetItem(
↪"parentHash")
                        SetItem("receiptTrie") SetItem("stateRoot") SetItem("timestamp
↪")
                        SetItem("transactionsTrie") SetItem("uncleHash")
                      )

    rule <k> check "blockHeader" : { "bloom"            : VALUE } => . ... </k>
↪<logsBloom>        VALUE </logsBloom>
    rule <k> check "blockHeader" : { "coinbase"         : VALUE } => . ... </k>
↪<coinbase>         VALUE </coinbase>
    rule <k> check "blockHeader" : { "difficulty"       : VALUE } => . ... </k>
↪<difficulty>       VALUE </difficulty>
    rule <k> check "blockHeader" : { "extraData"        : VALUE } => . ... </k>
↪<extraData>        VALUE </extraData>
    rule <k> check "blockHeader" : { "gasLimit"         : VALUE } => . ... </k>
↪<gasLimit>         VALUE </gasLimit>
    rule <k> check "blockHeader" : { "gasUsed"          : VALUE } => . ... </k>
↪<gasUsed>          VALUE </gasUsed>
```

(continues on next page)

(continued from previous page)

```
    rule <k> check "blockHeader" : { "mixHash"          : VALUE } => . ... </k>
↪<mixHash>          VALUE </mixHash>
    rule <k> check "blockHeader" : { "nonce"            : VALUE } => . ... </k>
↪<blockNonce>       VALUE </blockNonce>
    rule <k> check "blockHeader" : { "number"           : VALUE } => . ... </k>
↪<number>           VALUE </number>
    rule <k> check "blockHeader" : { "parentHash"       : VALUE } => . ... </k>
↪<previousHash>     VALUE </previousHash>
    rule <k> check "blockHeader" : { "receiptTrie"      : VALUE } => . ... </k>
↪<receiptsRoot>     VALUE </receiptsRoot>
    rule <k> check "blockHeader" : { "stateRoot"        : VALUE } => . ... </k>
↪<stateRoot>        VALUE </stateRoot>
    rule <k> check "blockHeader" : { "timestamp"        : VALUE } => . ... </k>
↪<timestamp>        VALUE </timestamp>
    rule <k> check "blockHeader" : { "transactionsTrie" : VALUE } => . ... </k>
↪<transactionsRoot> VALUE </transactionsRoot>
    rule <k> check "blockHeader" : { "uncleHash"        : VALUE } => . ... </k>
↪<ommersHash>       VALUE </ommersHash>

    rule <k> check "blockHeader" : { "hash": HASH:WordStack } => . ...</k>
         <previousHash>     HP </previousHash>
         <ommersHash>       HO </ommersHash>
         <coinbase>         HC </coinbase>
         <stateRoot>        HR </stateRoot>
         <transactionsRoot> HT </transactionsRoot>
         <receiptsRoot>     HE </receiptsRoot>
         <logsBloom>        HB </logsBloom>
         <difficulty>       HD </difficulty>
         <number>           HI </number>
         <gasLimit>         HL </gasLimit>
         <gasUsed>          HG </gasUsed>
         <timestamp>        HS </timestamp>
         <extraData>        HX </extraData>
         <mixHash>          HM </mixHash>
         <blockNonce>       HN </blockNonce>
      requires #blockHeaderHash(HP, HO, HC, HR, HT, HE, HB, HD, HI, HL, HG, HS, HX,␣
↪HM, HN) ==Int #asWord(HASH)

    rule check TESTID : { "genesisBlockHeader" : BLOCKHEADER } => check
↪"genesisBlockHeader" : BLOCKHEADER ~> failure TESTID
 // -----------------------------------------------------------------------------------
↪--------------------------------------
    rule <k> check "genesisBlockHeader" : { KEY : VALUE , REST } => check
↪"genesisBlockHeader" : { KEY : VALUE } ~> check "genesisBlockHeader" : { REST } ...
↪</k>
      requires REST =/=K .JSONList

    rule <k> check "genesisBlockHeader" : { KEY : VALUE } => .K ... </k> requires KEY␣
↪=/=String "hash"

    rule <k> check "genesisBlockHeader" : { "hash": (HASH:String => #asWord(
↪#parseByteStack(HASH))) } ... </k>
    rule <k> check "genesisBlockHeader" : { "hash": HASH } => . ... </k>
         <blockhash> ... ListItem(HASH) ListItem(_) </blockhash>

    rule <k> check TESTID : { "transactions" : TRANSACTIONS } => check "transactions"␣
↪: TRANSACTIONS ~> failure TESTID ... </k>
```

(continues on next page)

(continued from previous page)

```
 // ---------------------------------------------------------------------------------
↪-----------------------------------------
   rule <k> check "transactions" : [ .JSONList ] => . ... </k> <txOrder> .List      ␣
↪           </txOrder>
   rule <k> check "transactions" : { .JSONList } => . ... </k> <txOrder> ListItem(_)␣
↪=> .List ... </txOrder>

   rule <k> check "transactions" : [ TRANSACTION , REST ] => check "transactions" :␣
↪TRANSACTION   ~> check "transactions" : [ REST ] ... </k>
   rule <k> check "transactions" : { KEY : VALUE , REST } => check "transactions" :␣
↪(KEY : VALUE) ~> check "transactions" : { REST } ... </k>

   rule <k> check "transactions" : (KEY  : (VALUE:String    =>
↪#parseByteStack(VALUE))) ... </k>
   rule <k> check "transactions" : ("to" : (VALUE:WordStack => #asAccount(VALUE)))  ␣
↪   ... </k>
   rule <k> check "transactions" : (KEY  : (VALUE:WordStack => #padToWidth(32,␣
↪VALUE))) ... </k> requires KEY in (SetItem("r") SetItem("s")) andBool
↪#sizeWordStack(VALUE) <Int 32
   rule <k> check "transactions" : (KEY  : (VALUE:WordStack => #asWord(VALUE)))     ␣
↪   ... </k> requires KEY in (SetItem("gasLimit") SetItem("gasPrice") SetItem("nonce
↪") SetItem("v") SetItem("value"))

   rule <k> check "transactions" : ("data"     : VALUE) => . ... </k> <txOrder>␣
↪ListItem(TXID) ... </txOrder> <message> <msgID> TXID </msgID> <data>       VALUE </
↪data>        ... </message>
   rule <k> check "transactions" : ("gasLimit" : VALUE) => . ... </k> <txOrder>␣
↪ListItem(TXID) ... </txOrder> <message> <msgID> TXID </msgID> <txGasLimit> VALUE </
↪txGasLimit> ... </message>
   rule <k> check "transactions" : ("gasPrice" : VALUE) => . ... </k> <txOrder>␣
↪ListItem(TXID) ... </txOrder> <message> <msgID> TXID </msgID> <txGasPrice> VALUE </
↪txGasPrice> ... </message>
   rule <k> check "transactions" : ("nonce"    : VALUE) => . ... </k> <txOrder>␣
↪ListItem(TXID) ... </txOrder> <message> <msgID> TXID </msgID> <txNonce>    VALUE </
↪txNonce>    ... </message>
   rule <k> check "transactions" : ("r"        : VALUE) => . ... </k> <txOrder>␣
↪ListItem(TXID) ... </txOrder> <message> <msgID> TXID </msgID> <sigR>       VALUE </
↪sigR>        ... </message>
   rule <k> check "transactions" : ("s"        : VALUE) => . ... </k> <txOrder>␣
↪ListItem(TXID) ... </txOrder> <message> <msgID> TXID </msgID> <sigS>       VALUE </
↪sigS>        ... </message>
   rule <k> check "transactions" : ("to"       : VALUE) => . ... </k> <txOrder>␣
↪ListItem(TXID) ... </txOrder> <message> <msgID> TXID </msgID> <to>         VALUE </
↪to>          ... </message>
   rule <k> check "transactions" : ("v"        : VALUE) => . ... </k> <txOrder>␣
↪ListItem(TXID) ... </txOrder> <message> <msgID> TXID </msgID> <sigV>       VALUE </
↪sigV>        ... </message>
   rule <k> check "transactions" : ("value"    : VALUE) => . ... </k> <txOrder>␣
↪ListItem(TXID) ... </txOrder> <message> <msgID> TXID </msgID> <value>      VALUE </
↪value>       ... </message>
```

TODO: case with nonzero ommers.

```
   rule <k> check TESTID : { "uncleHeaders" : OMMERS } => check "ommerHeaders" :␣
↪OMMERS ~> failure TESTID ... </k>
 // ---------------------------------------------------------------------------------
↪-----------------------------
```

(continues on next page)

(continued from previous page)

```
    rule <k> check "ommerHeaders" : [ .JSONList ] => . ... </k> <ommerBlockHeaders> [␣
↪.JSONList ] </ommerBlockHeaders>
```

```
endmodule
```

CHAPTER

NINE

EDSL HIGH-LEVEL NOTATIONS

The eDSL high-level notations make the EVM specifications more succinct and closer to their high-level specifications. The succinctness increases the readability, and the closeness helps "eye-ball validation" of the specification refinement. The high-level notations are defined by translation to the corresponding EVM terms, and thus can be freely used with other EVM terms. The notations are inspired by the production compilers of the smart contract languages like Solidity and Vyper, and their definition is derived by formalizing the corresponding translation made by the compilers.

```
requires "evm.k"

module EDSL
    imports EVM
```

9.1 ABI Call Data

When a function is called in the EVM, its arguments are encoded in a single byte-array and put in the so-called 'call data' section. The encoding is defined in the Ethereum contract application binary interface (ABI) specification. The eDSL provides `#abiCallData`, a notation to specify the ABI call data in a way similar to a high-level function call notation, defined below. It specifies the function name and the (symbolic) arguments along with their types. For example, the following notation represents a data that encodes a call to the `transfer` function with two arguments: `TO`, the receiver account address of type `address` (an 160-bit unsigned integer), and `VALUE`, the value to transfer of type `unit256` (a 256-bit unsigned integer).

```
  #abiCallData("transfer", #address(TO), #uint256(VALUE))
```

which denotes (indeed, is translated to) the following byte array:

```
  F1 : F2 : F3 : F4 : T1 : ... : T32 : V1 : ... : V32
```

where `F1 : F2 : F3 : F4` is the (two's complement) byte-array representation of `2835717307`, the first four bytes of the hash value of the `transfer` function signature, `keccak256("transfer(address, unit256)")`, and `T1 : ... : T32` and `V1 : ... : V32` are the byte-array representations of `TO` and `VALUE` respectively.

```
    syntax TypedArg ::= #uint160 ( Int )
                      | #address ( Int )
                      | #uint256 ( Int )
                      | #int256  ( Int )
                      | #int128  ( Int )
                      | #bytes32 ( Int )
                      | #bool    ( Int )
                      | #bytes   ( Int , Int )
```

(continues on next page)

(continued from previous page)

```
// ------------------------------------------

   syntax TypedArgs ::= List{TypedArg, ","} [klabel(typedArgs)]
// ------------------------------------------------------------

   syntax WordStack ::= #abiCallData ( String , TypedArgs ) [function]
// -------------------------------------------------------------------
   rule #abiCallData( FNAME , ARGS )
     => #parseByteStack(substrString(Keccak256(#generateSignature(FNAME, ARGS)), 0,␣
↪8))
     ++ #encodeArgs(ARGS)

   syntax String ::= #generateSignature     ( String, TypedArgs ) [function]
                   | #generateSignatureArgs ( TypedArgs )         [function]
// --------------------------------------------------------------------------
   rule #generateSignature( FNAME , ARGS ) => FNAME +String "(" +String
↪#generateSignatureArgs(ARGS) +String ")"

   rule #generateSignatureArgs(.TypedArgs)                            => ""
   rule #generateSignatureArgs(TARGA:TypedArg, .TypedArgs)            =>
↪#typeName(TARGA)
   rule #generateSignatureArgs(TARGA:TypedArg, TARGB:TypedArg, TARGS) =>
↪#typeName(TARGA) +String "," +String #generateSignatureArgs(TARGB, TARGS)

   syntax String ::= #typeName ( TypedArg ) [function]
// ---------------------------------------------------
   rule #typeName(#uint160( _ )) => "uint160"
   rule #typeName(#address( _ )) => "address"
   rule #typeName(#uint256( _ )) => "uint256"
   rule #typeName( #int256( _ )) => "int256"
   rule #typeName( #int128( _ )) => "int128"
   rule #typeName(#bytes32( _ )) => "bytes32"
   rule #typeName(   #bool( _ )) => "bool"
   rule #typeName( #bytes(_, _)) => "bytes"

   syntax WordStack ::= #encodeArgs    ( TypedArgs )                              ␣
↪[function]
   syntax WordStack ::= #encodeArgsAux ( TypedArgs , Int , WordStack , WordStack )␣
↪[function]
// ------------------------------------------------------------------------------
↪--------
   rule #encodeArgs(ARGS) => #encodeArgsAux(ARGS, #lenOfHeads(ARGS), .WordStack, .
↪WordStack)

   rule #encodeArgsAux(.TypedArgs, _:Int, HEADS, TAILS) => HEADS ++ TAILS

   rule #encodeArgsAux((ARG, ARGS), OFFSET, HEADS, TAILS)
       => #encodeArgsAux(ARGS, OFFSET, HEADS ++ #enc(ARG), TAILS)
     requires #isStaticType(ARG)

   rule #encodeArgsAux((ARG, ARGS), OFFSET, HEADS, TAILS)
       => #encodeArgsAux(ARGS, OFFSET +Int #sizeOfDynamicType(ARG), HEADS ++ #enc(
↪#uint256(OFFSET)), TAILS ++ #enc(ARG))
     requires notBool(#isStaticType(ARG))

   syntax Int ::= #lenOfHeads ( TypedArgs ) [function]
// ---------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
    rule #lenOfHeads(.TypedArgs) => 0
    rule #lenOfHeads(ARG, ARGS)  => #lenOfHead(ARG) +Int #lenOfHeads(ARGS)

    syntax Int ::= #lenOfHead ( TypedArg ) [function]
 // ------------------------------------------------
    rule #lenOfHead(#uint160( _ )) => 32
    rule #lenOfHead(#address( _ )) => 32
    rule #lenOfHead(#uint256( _ )) => 32
    rule #lenOfHead( #int256( _ )) => 32
    rule #lenOfHead( #int128( _ )) => 32
    rule #lenOfHead(#bytes32( _ )) => 32
    rule #lenOfHead(   #bool( _ )) => 32
    rule #lenOfHead( #bytes(_, _)) => 32

    syntax Bool ::= #isStaticType ( TypedArg ) [function]
 // ----------------------------------------------------
    rule #isStaticType(#uint160( _ )) => true
    rule #isStaticType(#address( _ )) => true
    rule #isStaticType(#uint256( _ )) => true
    rule #isStaticType( #int256( _ )) => true
    rule #isStaticType( #int128( _ )) => true
    rule #isStaticType(#bytes32( _ )) => true
    rule #isStaticType(   #bool( _ )) => true
    rule #isStaticType( #bytes(_, _)) => false

    syntax Int ::= #sizeOfDynamicType ( TypedArg ) [function]
 // --------------------------------------------------------
    rule #sizeOfDynamicType(#bytes(N, _)) => 32 +Int #ceil32(N)

    syntax WordStack ::= #enc ( TypedArg ) [function]
 // ------------------------------------------------
    // static Type
    rule #enc(#uint160( DATA )) => #padToWidth(32, #asByteStack(#getValue(#uint160(␣
↪DATA ))))
    rule #enc(#address( DATA )) => #padToWidth(32, #asByteStack(#getValue(#address(␣
↪DATA ))))
    rule #enc(#uint256( DATA )) => #padToWidth(32, #asByteStack(#getValue(#uint256(␣
↪DATA ))))
    rule #enc( #int256( DATA )) => #padToWidth(32, #asByteStack(#getValue( #int256(␣
↪DATA ))))
    rule #enc( #int128( DATA )) => #padToWidth(32, #asByteStack(#getValue( #int128(␣
↪DATA ))))
    rule #enc(#bytes32( DATA )) => #padToWidth(32, #asByteStack(#getValue(#bytes32(␣
↪DATA ))))
    rule #enc(   #bool( DATA )) => #padToWidth(32, #asByteStack(#getValue(   #bool(␣
↪DATA ))))

    // dynamic Type
    rule #enc( #bytes(N, DATA)) => #enc(#uint256(N)) ++ #padToWidth(#ceil32(N),
↪#asByteStack(DATA))

    syntax Int ::= #getValue ( TypedArg ) [function]
 // -----------------------------------------------
    rule #getValue(#uint160( DATA )) => DATA
      requires minUInt160 <=Int DATA andBool DATA <=Int maxUInt160

    rule #getValue(#address( DATA )) => DATA
```

(continues on next page)

(continued from previous page)

```
      requires minUInt160 <=Int DATA andBool DATA <=Int maxUInt160

    rule #getValue(#uint256( DATA )) => DATA
      requires minUInt256 <=Int DATA andBool DATA <=Int maxUInt256

    rule #getValue( #int256( DATA )) => #unsigned(DATA)
      requires minSInt256 <=Int DATA andBool DATA <=Int maxSInt256

    rule #getValue( #int128( DATA )) => #unsigned(DATA)
      requires minSInt128 <=Int DATA andBool DATA <=Int maxSInt128

    rule #getValue(#bytes32( DATA )) => DATA
      requires minUInt256 <=Int DATA andBool DATA <=Int maxUInt256

    rule #getValue(   #bool( DATA )) => DATA
      requires 0 <=Int DATA andBool DATA <=Int 1

    syntax Int ::= #ceil32 ( Int ) [function]
 // -----------------------------------------
    rule #ceil32(N) => ((N +Int 31) /Int 32) *Int 32
```

9.2 ABI Event Logs

EVM logs are special data structures in the blockchain, being searchable by off-chain clients. Events are high-level wrappers of the EVM logs provided in the high-level languages. Contracts can declare and generate the events, which will be compiled down to the EVM bytecode using the EVM log instructions. The encoding scheme of the events in the EVM logs is defined in the Ethereum contract application binary interface (ABI) specification, leveraging the ABI call data encoding scheme.

The eDSL provides `#abiEventLog`, a notation to specify the EVM logs in the high-level events, defined below. It specifies the contract account address, the event name, and the event arguments. For example, the following notation represents an EVM log data that encodes the `Transfer` event generated by the `transfer` function, where `ACCT_ID` is the account address, and `CALLER_ID`, `TO_ID`, and `VALUE` are the event arguments. Each argument is tagged with its ABI type (`#address` or `#uint256`), and the `indexed` attribute (`#indexed`) if any, according to the event declaration in the contract.

```
  #abiEventLog(ACCT_ID, "Transfer", #indexed(#address(CALLER_ID)), #indexed(
↪#address(TO_ID)), #uint256(VALUE))
```

The above notation denotes (i.e., is translated to) the following EVM log data structure:

```
  { ACCT_ID                                                                 ␣
↪                                                                           ␣
↪                                                                          ␣
↪|  |`
  | 100389287136786176327247604509743168900146139575972864366142685224231313322991
  : CALLER_ID                                                               ␣
↪                                                                           ␣
↪                                                                          ␣
↪|/|
  : TO_ID                                                                   ␣
↪                                                                           ␣
↪                                                                          ␣
↪|  |
```

(continues on next page)

(continued from previous page)

```
  : .WordStack                                                                    ␣
↪                                                                                 ␣
↪                                                                                ␣
↪| |
  | #asByteStackInWidth(VALUE, 32)                                                ␣
↪                                                                                 ␣
↪                                                                                ␣
↪| |
  }
```

where 100389287136786176327247604509743168900146139575972864366142685224231313322991 is the hash value of the event signature, keccak256("Transfer(address,address,unit256)").

```
   syntax EventArg ::= TypedArg
                     | #indexed ( TypedArg )
 // -----------------------------------------

   syntax EventArgs ::= List{EventArg, ","} [klabel(eventArgs)]
 // -------------------------------------------------------------

   syntax SubstateLogEntry ::= #abiEventLog ( Int , String , EventArgs ) [function]
 // --------------------------------------------------------------------------------
   rule #abiEventLog(ACCT_ID, EVENT_NAME, EVENT_ARGS)
     => { ACCT_ID | #getEventTopics(EVENT_NAME, EVENT_ARGS) | #getEventData(EVENT_
↪ARGS) }

   syntax List ::= #getEventTopics ( String , EventArgs ) [function]
 // ------------------------------------------------------------------
   rule #getEventTopics(ENAME, EARGS)
     => ListItem(#parseHexWord(Keccak256(#generateSignature(ENAME,
↪#getTypedArgs(EARGS)))))
        #getIndexedArgs(EARGS)

   syntax TypedArgs ::= #getTypedArgs ( EventArgs ) [function]
 // ------------------------------------------------------------
   rule #getTypedArgs(#indexed(E), ES) => E, #getTypedArgs(ES)
   rule #getTypedArgs(E:TypedArg,  ES) => E, #getTypedArgs(ES)
   rule #getTypedArgs(.EventArgs)      => .TypedArgs

   syntax List ::= #getIndexedArgs ( EventArgs ) [function]
 // ---------------------------------------------------------
   rule #getIndexedArgs(#indexed(E), ES) => ListItem(#getValue(E))
↪#getIndexedArgs(ES)
   rule #getIndexedArgs(_:TypedArg,  ES) =>
↪#getIndexedArgs(ES)
   rule #getIndexedArgs(.EventArgs)      => .List

   syntax WordStack ::= #getEventData ( EventArgs ) [function]
 // ------------------------------------------------------------
   rule #getEventData(#indexed(_), ES) =>            #getEventData(ES)
   rule #getEventData(E:TypedArg,  ES) => #enc(E) ++ #getEventData(ES)
   rule #getEventData(.EventArgs)      => .WordStack
```

9.3 Hashed Location for Storage

The storage accommodates permanent data such as the `balances` map. A map is laid out in the storage where the map entries are scattered over the entire storage space using the (256-bit) hash of each key to determine the location. The detailed mechanism of calculating the location varies by compilers. In Vyper, for example, `map[key1][key2]` is stored at the location:

```
hash(hash(idx(map)) + key1) + key2
```

where `idx(map)` is the position index of `map` in the program, and `+` is the addition modulo `2**256`, while in Solidity, it is stored at:

```
hash(key2 ++ hash(key1 ++ idx(map)))
```

where `++` is byte-array concatenation.

The eDSL provides `#hashedLocation` that allows to uniformly specify the locations in a form parameterized by the underlying compilers. For example, the location of `map[key1][key2]` can be specified as follows, where `{COMPILER}` is a place-holder to be replaced by the name of the compiler. Note that the keys are separated by the white spaces instead of commas.

```
#hashedLocation({COMPILER}, idx(map), key1 key2)
```

This notation makes the specification independent of the underlying compilers, enabling it to be reused for differently compiled programs. Specifically, `#hashedLocation` is defined as follows, capturing the storage layout schemes of Solidity and Vyper.

```
    syntax IntList ::= List{Int, ""}                             [klabel(intList)]
    syntax Int     ::= #hashedLocation( String , Int , IntList ) [function]
 // -----------------------------------------------------------------------
    rule #hashedLocation(LANG, BASE, .IntList) => BASE

    rule #hashedLocation("Vyper",    BASE, OFFSET OFFSETS) => #hashedLocation("Vyper",
↪     keccakIntList(BASE) +Word OFFSET, OFFSETS)
    rule #hashedLocation("Solidity", BASE, OFFSET OFFSETS) => #hashedLocation(
↪"Solidity", keccakIntList(OFFSET BASE),       OFFSETS)

    syntax Int ::= keccakIntList( IntList ) [function]
 // --------------------------------------------------
    rule keccakIntList(VS) => keccak(intList2ByteStack(VS))

    syntax WordStack ::= intList2ByteStack( IntList ) [function]
 // ------------------------------------------------------------
    rule intList2ByteStack(.IntList) => .WordStack
    rule intList2ByteStack(V VS)     => #padToWidth(32, #asByteStack(V)) ++␣
↪intList2ByteStack(VS)
      requires 0 <=Int V andBool V <Int pow256
endmodule
```

CHAPTER

TEN

NETWORK STATE

This file represents all the network state present in the EVM. It will incrementally build up to supporting the entire EVM-C API.

```
module NETWORK
```

10.1 EVM Status Codes

10.1.1 Exceptional Codes

The following codes all indicate that the VM ended execution with an exception, but give details about how.

- `EVMC_FAILURE` is a catch-all for generic execution failure.
- `EVMC_INVALID_INSTRUCTION` indicates reaching the designated `INVALID` opcode.
- `EVMC_UNDEFINED_INSTRUCTION` indicates that an undefined opcode has been reached.
- `EVMC_OUT_OF_GAS` indicates that execution exhausted the gas supply.
- `EVMC_BAD_JUMP_DESTINATION` indicates a `JUMP*` to a non-`JUMPDEST` location.
- `EVMC_STACK_OVERFLOW` indicates pushing more than 1024 elements onto the wordstack.
- `EVMC_STACK_UNDERFLOW` indicates popping elements off an empty wordstack.
- `EVMC_CALL_DEPTH_EXCEEDED` indicates that we have executed too deeply a nested sequence of `CALL*` or `CREATE` opcodes.
- `EVMC_INVALID_MEMORY_ACCESS` indicates that a bad memory access occured. This can happen when accessing local memory with `CODECOPY*` or `CALLDATACOPY`, or when accessing return data with `RETURNDATACOPY`.
- `EVMC_STATIC_MODE_VIOLATION` indicates that a `STATICCALL` tried to change state. **TODO:** Avoid `_ERROR` suffix that suggests fatal error.
- `EVMC_PRECOMPILE_FAILURE` indicates an errors in the precompiled contracts (eg. invalid points handed to elliptic curve functions).

```
    syntax ExceptionalStatusCode ::= "EVMC_FAILURE"
                                   | "EVMC_INVALID_INSTRUCTION"
                                   | "EVMC_UNDEFINED_INSTRUCTION"
                                   | "EVMC_OUT_OF_GAS"
                                   | "EVMC_BAD_JUMP_DESTINATION"
                                   | "EVMC_STACK_OVERFLOW"
```

(continues on next page)

(continued from previous page)

```
                                            | "EVMC_STACK_UNDERFLOW"
                                            | "EVMC_CALL_DEPTH_EXCEEDED"
                                            | "EVMC_INVALID_MEMORY_ACCESS"
                                            | "EVMC_STATIC_MODE_VIOLATION"
                                            | "EVMC_PRECOMPILE_FAILURE"
```

10.1.2 Ending Codes

These additional status codes indicate that execution has ended in some non-exceptional way.

- `EVMC_SUCCESS` indicates successful end of execution.
- `EVMC_REVERT` indicates that the contract called `REVERT`.

```
    syntax EndStatusCode ::= ExceptionalStatusCode
                           | "EVMC_SUCCESS"
                           | "EVMC_REVERT"
```

10.1.3 Other Codes

The following codes indicate other non-execution errors with the VM.

- `EVMC_REJECTED` indicates malformed or wrong-version EVM bytecode.
- `EVMC_INTERNAL_ERROR` indicates some other error that is unrecoverable but not due to the bytecode.
- `.StatusCode` is an extra code added for "unset or unknown".

```
    syntax StatusCode ::= EndStatusCode
                        | "EVMC_REJECTED"
                        | "EVMC_INTERNAL_ERROR"
                        | ".StatusCode"
```

10.2 Client/Network Codes

The following are status codes used to report network state failures to the EVM from the client. These are not present in the EVM-C API.

- `EVMC_ACCOUNT_ALREADY_EXISTS` indicates that a newly created account already exists.
- `EVMC_BALANCE_UNDERFLOW` indicates an attempt to create an account which already exists.

```
    syntax ExceptionalStatusCode ::= "EVMC_ACCOUNT_ALREADY_EXISTS"
                                   | "EVMC_BALANCE_UNDERFLOW"
```

```
endmodule
```

CHAPTER

ELEVEN

RESOURCES

CHAPTER

TWELVE

EVM WORDS

12.1 Module EVM-DATA

EVM uses bounded 256 bit integer words, and sometimes also bytes (8 bit words). Here we provide the arithmetic of these words, as well as some data-structures over them. Both are implemented using K's `Int`.

```
requires "krypto.k"

module EVM-DATA
    imports KRYPTO
    imports STRING-BUFFER
    imports DOMAINS

    syntax KResult ::= Int
```

12.2 JSON Formatting

The JSON format is used extensively for communication in the Ethereum circles. Writing a JSON-ish parser in K takes 6 lines.

```
    syntax JSONList ::= List{JSON,","}
    syntax JSONKey  ::= String | Int
    syntax JSON     ::= String
                      | JSONKey ":" JSON
                      | "{" JSONList "}"
                      | "[" JSONList "]"
 // ------------------------------------
```

12.3 Utilities

12.3.1 Important Powers

Some important numbers that are referred to often during execution. These can be used for pattern-matching on the LHS of rules as well (`macro` attribute expands all occurances of these in rules).

```
    syntax Int ::= "pow256" [function] /* 2 ^Int 256 */
                 | "pow255" [function] /* 2 ^Int 255 */
                 | "pow160" [function] /* 2 ^Int 160 */
```

(continues on next page)

(continued from previous page)

```
                | "pow16"  [function] /* 2 ^Int 16  */
// --------------------------------------------------
   rule pow256 =>␣
↪115792089237316195423570985008687907853269984665640564039457584007913129639936␣
↪[macro]
   rule pow255 =>␣
↪57896044618658097711785492504343953926634992332820282019728792003956564819968 ␣
↪[macro]
   rule pow160 => 1461501637330902918203684832716283019655932542976 [macro]
   rule pow16  => 65536 [macro]

   syntax Int ::= "minSInt128"      [function]
                | "maxSInt128"      [function]
                | "minUInt128"      [function]
                | "maxUInt128"      [function]
                | "minUInt160"      [function]
                | "maxUInt160"      [function]
                | "minSInt256"      [funciton]
                | "maxSInt256"      [function]
                | "minUInt256"      [function]
                | "maxUInt256"      [function]
                | "minSFixed128x10" [funciton]
                | "maxSFixed128x10" [function]
                | "minUFixed128x10" [function]
                | "maxUFixed128x10" [function]
// -------------------------------------------
   rule minSInt128      => -170141183460469231731687303715884105728                ␣
↪                       [macro]  /*  -2^127      */
   rule maxSInt128      =>  170141183460469231731687303715884105727                ␣
↪                       [macro]  /*   2^127 - 1  */
   rule minSFixed128x10 => -1701411834604692317316873037158841057280000000000      ␣
↪                       [macro]  /* (-2^127    ) * 10^10 */
   rule maxSFixed128x10 =>  1701411834604692317316873037158841057270000000000      ␣
↪                       [macro]  /* ( 2^127 - 1) * 10^10 */
   rule minSInt256      => -
↪57896044618658097711785492504343953926634992332820282019728792003956564819968 ␣
↪[macro]  /*  -2^255      */
   rule maxSInt256      => ␣
↪57896044618658097711785492504343953926634992332820282019728792003956564819967 ␣
↪[macro]  /*   2^255 - 1  */

   rule minUInt128      =>  0                                                      ␣
↪                       [macro]
   rule maxUInt128      =>  340282366920938463463374607431768211455                ␣
↪                       [macro]  /*   2^128 - 1  */
   rule minUFixed128x10 =>  0                                                      ␣
↪                       [macro]
   rule maxUFixed128x10 =>  3402823669209384634633746074317682114550000000000      ␣
↪                       [macro]  /* ( 2^128 - 1) * 10^10 */
   rule minUInt160      =>  0                                                      ␣
↪                       [macro]
   rule maxUInt160      =>  1461501637330902918203684832716283019655932542975      ␣
↪                       [macro]  /*   2^160 - 1  */
   rule minUInt256      =>  0                                                      ␣
↪                       [macro]
   rule maxUInt256      => ␣
↪115792089237316195423570985008687907853269984665640564039457584007913129639935␣
↪[macro]  /*   2^256 - 1  */
```

(continues on next page)

(continued from previous page)

- Range of types

```
    syntax Bool ::= #rangeSInt     ( Int , Int )        [function]
                  | #rangeUInt     ( Int , Int )        [function]
                  | #rangeSFixed   ( Int , Int , Int )  [function]
                  | #rangeUFixed   ( Int , Int , Int )  [function]
                  | #rangeAddress  ( Int )              [function]
                  | #rangeBytes    ( Int , Int )        [function]
 // --------------------------------------------------------------
    rule #rangeSInt    ( 128 ,       X ) => #range ( minSInt128       <= X <=␣
↪maxSInt128       ) [macro]
    rule #rangeSInt    ( 256 ,       X ) => #range ( minSInt256       <= X <=␣
↪maxSInt256       ) [macro]
    rule #rangeUInt    ( 128 ,       X ) => #range ( minUInt128       <= X <=␣
↪maxUInt128       ) [macro]
    rule #rangeUInt    ( 256 ,       X ) => #range ( minUInt256       <= X <=␣
↪maxUInt256       ) [macro]
    rule #rangeSFixed  ( 128 , 10 , X ) => #range ( minSFixed128x10 <= X <=␣
↪maxSFixed128x10 ) [macro]
    rule #rangeUFixed  ( 128 , 10 , X ) => #range ( minUFixed128x10 <= X <=␣
↪maxUFixed128x10 ) [macro]
    rule #rangeAddress (             X ) => #range ( minUInt160       <= X <=␣
↪maxUInt160       ) [macro]
    rule #rangeBytes   (  32 ,       X ) => #range ( minUInt256       <= X <=␣
↪maxUInt256       ) [macro]

    syntax Bool ::= "#range" "(" Int "<"  Int "<"  Int ")" [function]
                  | "#range" "(" Int "<"  Int "<=" Int ")" [function]
                  | "#range" "(" Int "<=" Int "<"  Int ")" [function]
                  | "#range" "(" Int "<=" Int "<=" Int ")" [function]
 // -------------------------------------------------------------------
    rule #range ( LB <  X <  UB ) => LB  <Int X andBool X  <Int UB [macro]
    rule #range ( LB <  X <= UB ) => LB  <Int X andBool X <=Int UB [macro]
    rule #range ( LB <= X <  UB ) => LB <=Int X andBool X  <Int UB [macro]
    rule #range ( LB <= X <= UB ) => LB <=Int X andBool X <=Int UB [macro]
```

- `chop` interperets an integer modulo $2^256$.

```
    syntax Int ::= chop ( Int ) [function, smtlib(chop)]
 // ----------------------------------------------------
    rule chop ( I:Int ) => I modInt pow256 [concrete, smt-lemma]
```

12.3.2 Boolean Conversions

Primitives provide the basic conversion from K's sorts `Int` and `Bool` to EVM's words.

- `bool2Word` interperets a `Bool` as a `Int`.
- `word2Bool` interperets a `Int` as a `Bool`.

```
    syntax Int ::= bool2Word ( Bool ) [function]
 // --------------------------------------------
    rule bool2Word( B:Bool ) => 1 requires B
    rule bool2Word( B:Bool ) => 0 requires notBool B
```

(continues on next page)

(continued from previous page)

```
    syntax Bool ::= word2Bool ( Int ) [function]
 // --------------------------------------------
    rule word2Bool( 0 ) => false
    rule word2Bool( W ) => true  requires W =/=K 0
```

- `sgn` gives the twos-complement interperetation of the sign of a word.
- `abs` gives the twos-complement interperetation of the magnitude of a word.

```
    syntax Int ::= sgn ( Int ) [function]
                 | abs ( Int ) [function]
 // -------------------------------------
    rule sgn(I) => -1 requires I >=Int pow255
    rule sgn(I) => 1  requires I <Int pow255

    rule abs(I) => 0 -Word I requires sgn(I) ==K -1
    rule abs(I) => I         requires sgn(I) ==K 1
```

- #signed : uInt256 -> sInt256 (i.e., [minUInt256..maxUInt256] -> [minSInt256..maxSInt256])
- #unsigned : sInt256 -> uInt256 (i.e., [minSInt256..maxSInt256] -> [minUInt256..maxUInt256])

```
    syntax Int ::= #signed ( Int ) [function]
 // -----------------------------------------
    rule #signed(DATA) => DATA
      requires 0 <=Int DATA andBool DATA <=Int maxSInt256

    rule #signed(DATA) => DATA -Int pow256
      requires maxSInt256 <Int DATA andBool DATA <=Int maxUInt256

    syntax Int ::= #unsigned ( Int ) [function]
 // -----------------------------------------
    rule #unsigned(DATA) => DATA
      requires 0 <=Int DATA andBool DATA <=Int maxSInt256

    rule #unsigned(DATA) => pow256 +Int DATA
      requires minSInt256 <=Int DATA andBool DATA <Int 0
```

12.3.3 Empty Account

- `.Account` represents the case when an account ID is referenced in the yellowpaper, but the actual value of the account ID is the empty set. This is used, for example, when referring to the destination of a message which creates a new contract.

```
    syntax Account ::= ".Account" | Int
```

12.3.4 Symbolic Words

- `#symbolicWord` generates a fresh existentially-bound symbolic word.

```
    syntax Int ::= "#symbolicWord" [function]
 // -----------------------------------------
    rule #symbolicWord => chop ( ?X:Int )
```

12.4 Word Operations

12.4.1 Low-Level

- `up/Int` performs integer division but rounds up instead of down.

NOTE: Here, we choose to add `I2 -Int 1` to the numerator beforing doing the division to mimic the C++ implementation. You could alternatively calculate `I1 modInt I2`, then add one to the normal integer division afterward depending on the result.

```
    syntax Int ::= Int "up/Int" Int [function]
 // ------------------------------------------
    rule I1 up/Int 0  => 0
    rule I1 up/Int 1  => I1
    rule I1 up/Int I2 => (I1 +Int (I2 -Int 1)) /Int I2 requires I2 >Int 1
```

- `log256Int` returns the log base 256 (floored) of an integer.

```
    syntax Int ::= log256Int ( Int ) [function]
 // -------------------------------------------
    rule log256Int(N) => log2Int(N) /Int 8
```

The corresponding `<op>Word` operations automatically perform the correct modulus for EVM words.

```
    syntax Int ::= Int "+Word" Int [function]
                 | Int "*Word" Int [function]
                 | Int "-Word" Int [function]
                 | Int "/Word" Int [function]
                 | Int "%Word" Int [function]
 // -----------------------------------------
    rule W0 +Word W1 => chop( W0 +Int W1 )
    rule W0 -Word W1 => chop( W0 -Int W1 ) requires W0 >=Int W1
    rule W0 -Word W1 => chop( (W0 +Int pow256) -Int W1 ) requires W0 <Int W1
    rule W0 *Word W1 => chop( W0 *Int W1 )
    rule W0 /Word 0  => 0
    rule W0 /Word W1 => chop( W0 /Int W1 ) requires W1 =/=K 0
    rule W0 %Word 0  => 0
    rule W0 %Word W1 => chop( W0 modInt W1 ) requires W1 =/=K 0
```

Care is needed for `^Word` to avoid big exponentiation. The helper `powmod` is a totalization of the operator `_^%Int__` (which comes with K). `_^%Int__` is not defined when the modulus (third argument) is zero, but `powmod` is.

```
    syntax Int ::= Int "^Word" Int       [function]
    syntax Int ::= powmod(Int, Int, Int) [function]
 // -----------------------------------------------
    rule W0 ^Word W1 => powmod(W0, W1, pow256)

    rule powmod(W0, W1, W2) => W0 ^%Int W1 W2 requires W2 =/=Int 0
    rule powmod(W0, W1, 0)  => 0
```

`/sWord` and `%sWord` give the signed interperetations of `/Word` and `%Word`.

```
    syntax Int ::= Int "/sWord" Int [function]
                 | Int "%sWord" Int [function]
 // ------------------------------------------
    rule W0 /sWord W1 => #sgnInterp(sgn(W0) *Int sgn(W1) , abs(W0) /Word abs(W1))
    rule W0 %sWord W1 => #sgnInterp(sgn(W0)              , abs(W0) %Word abs(W1))
```

(continues on next page)

(continued from previous page)

```
    syntax Int ::= #sgnInterp ( Int , Int ) [function]
 // --------------------------------------------------
    rule #sgnInterp( 0  , W1 ) => 0
    rule #sgnInterp( W0 , W1 ) => W1          requires W0 >Int 0
    rule #sgnInterp( W0 , W1 ) => 0 -Word W1 requires W0 <Int 0
```

12.4.2 Word Comparison

The `<op>Word` comparisons similarly lift K operators to EVM ones:

```
    syntax Int ::= Int "<Word"  Int [function]
                 | Int ">Word"  Int [function]
                 | Int "<=Word" Int [function]
                 | Int ">=Word" Int [function]
                 | Int "==Word" Int [function]
 // ------------------------------------------
    rule W0 <Word  W1 => bool2Word(W0 <Int  W1)
    rule W0 >Word  W1 => bool2Word(W0 >Int  W1)
    rule W0 <=Word W1 => bool2Word(W0 <=Int W1)
    rule W0 >=Word W1 => bool2Word(W0 >=Int W1)
    rule W0 ==Word W1 => bool2Word(W0 ==Int W1)
```

- `s<Word` implements a less-than for `Word` (with signed interperetation).

```
    syntax Int ::= Int "s<Word" Int [function]
 // ------------------------------------------
    rule W0 s<Word W1 => W0 <Word W1           requires sgn(W0) ==K 1  andBool␣
↪sgn(W1) ==K 1
    rule W0 s<Word W1 => bool2Word(false)      requires sgn(W0) ==K 1  andBool␣
↪sgn(W1) ==K -1
    rule W0 s<Word W1 => bool2Word(true)       requires sgn(W0) ==K -1 andBool␣
↪sgn(W1) ==K 1
    rule W0 s<Word W1 => abs(W1) <Word abs(W0) requires sgn(W0) ==K -1 andBool␣
↪sgn(W1) ==K -1
```

12.4.3 Bitwise Operators

Bitwise logical operators are lifted from the integer versions.

```
    syntax Int ::= "~Word" Int        [function]
                 | Int "|Word"   Int [function]
                 | Int "&Word"   Int [function]
                 | Int "xorWord" Int [function]
 // -------------------------------------------
    rule ~Word W        => chop( W xorInt (pow256 -Int 1) )
    rule W0 |Word   W1 => chop( W0 |Int W1 )
    rule W0 &Word   W1 => chop( W0 &Int W1 )
    rule W0 xorWord W1 => chop( W0 xorInt W1 )
```

- `bit` gets bit $N$ (0 being MSB).
- `byte` gets byte $N$ (0 being the MSB).

```
    syntax Int ::= bit  ( Int , Int ) [function]
                 | byte ( Int , Int ) [function]
 // ---------------------------------------------
    rule bit(N, _)  => 0 requires N <Int 0 orBool N >=Int 256
    rule byte(N, _) => 0 requires N <Int 0 orBool N >=Int 32

    rule bit(N, W)  => (W >>Int (255 -Int N)) modInt 2                  requires N␣
↪>=Int 0 andBool N <Int 256
    rule byte(N, W) => (W >>Int (256 -Int (8 *Int (N +Int 1)))) modInt 256 requires N␣
↪>=Int 0 andBool N <Int 32
```

- `#nBits` shifts in $N$ ones from the right.
- `#nBytes` shifts in $N$ bytes of ones from the right.
- `_<<Byte_` shifts an integer 8 bits to the left.

```
    syntax Int ::= #nBits  ( Int )   [function]
                 | #nBytes ( Int )   [function]
                 | Int "<<Byte" Int [function]
 // ------------------------------------------
    rule #nBits(N)  => (1 <<Int N) -Int 1  requires N >=Int 0
    rule #nBytes(N) => #nBits(N *Int 8)    requires N >=Int 0
    rule N <<Byte M => N <<Int (8 *Int M)
```

- `signextend(N, W)` sign-extends from byte $N$ of $W$ (0 being MSB).

```
    syntax Int ::= signextend( Int , Int ) [function]
 // -------------------------------------------------
    rule signextend(N, W) => W requires N >=Int 32 orBool N <Int 0
    rule signextend(N, W) => chop( (#nBytes(31 -Int N) <<Byte (N +Int 1)) |Int W )␣
↪requires N <Int 32 andBool N >=Int 0 andBool         word2Bool(bit(256 -Int (8 *Int␣
↪(N +Int 1)), W))
    rule signextend(N, W) => chop( #nBytes(N +Int 1)                       &Int W )␣
↪requires N <Int 32 andBool N >=Int 0 andBool notBool word2Bool(bit(256 -Int (8 *Int␣
↪(N +Int 1)), W))
```

- `keccak` serves as a wrapper around the `Keccak256` in `KRYPTO`.

```
    syntax Int ::= keccak ( WordStack ) [function, smtlib(smt_keccak)]
 // ------------------------------------------------------------------
    rule keccak(WS) => #parseHexWord(Keccak256(#unparseByteStack(WS))) [concrete]
```

CHAPTER

THIRTEEN

DATA-STRUCTURES OVER WORD

13.1 A WordStack for EVM

13.1.1 As a cons-list

A cons-list is used for the EVM wordstack.

- `.WordStack` serves as the empty worstack, and
- `_:_` serves as the "cons" operator.

```
   syntax WordStack [flatPredicate]
   syntax WordStack ::= ".WordStack" | Int ":" WordStack
// -----------------------------------------------------
```

- `_++_` acts as `WordStack` append.
- `#take(N , WS)` keeps the first $N$ elements of a `WordStack` (passing with zeros as needed).
- `#drop(N , WS)` removes the first $N$ elements of a `WordStack`.

```
   syntax WordStack ::= WordStack "++" WordStack [function, right]
// ---------------------------------------------------------------
   rule .WordStack ++ WS' => WS'
   rule (W : WS)   ++ WS' => W : (WS ++ WS')

   syntax WordStack ::= #take ( Int , WordStack ) [function]
// ---------------------------------------------------------
   rule #take(0, WS)           => .WordStack
   rule #take(N, .WordStack) => 0 : #take(N -Int 1, .WordStack) requires N >Int 0
   rule #take(N, (W : WS))    => W : #take(N -Int 1, WS)          requires N >Int 0

   syntax WordStack ::= #drop ( Int , WordStack ) [function]
// ---------------------------------------------------------
   rule #drop(0, WS)           => WS
   rule #drop(N, .WordStack) => .WordStack
   rule #drop(N, (W : WS))    => #drop(N -Int 1, WS) requires N >Int 0
```

13.1.2 Element Access

- `WS [ N ]` accesses element $N$ of $WS$.
- `WS [ N .. W ]` access the range of `WS` beginning with `N` of width `W`.

- `WS [ N := W ]` sets element $N$ of $WS$ to $W$ (padding with zeros as needed).

```
    syntax Int ::= WordStack "[" Int "]" [function]
 // -----------------------------------------------
    rule (W0 : WS)    [0] => W0
    rule (.WordStack)[N] => 0             requires N >Int 0
    rule (W0 : WS)    [N] => WS[N -Int 1] requires N >Int 0

    syntax WordStack ::= WordStack "[" Int ".." Int "]" [function]
 // --------------------------------------------------------------
    rule WS [ START .. WIDTH ] => #take(WIDTH, #drop(START, WS))

    syntax WordStack ::= WordStack "[" Int ":=" Int "]" [function]
 // --------------------------------------------------------------
    rule (W0 : WS)  [ 0 := W ] => W  : WS
    rule .WordStack [ N := W ] => 0  : (.WordStack [ N -Int 1 := W ]) requires N >Int␣
↪0
    rule (W0 : WS)  [ N := W ] => W0 : (WS [ N -Int 1 := W ])         requires N >Int␣
↪0
```

- `#sizeWordStack` calculates the size of a `WordStack`.
- `_in_` determines if a `Int` occurs in a `WordStack`.

```
    syntax Int ::= #sizeWordStack ( WordStack )       [function,␣
↪smtlib(sizeWordStack)]
                 | #sizeWordStack ( WordStack , Int ) [function,␣
↪klabel(sizeWordStackAux), smtlib(sizeWordStackAux)]
 // ---------------------------------------------------------------------------------
↪------------------------------
    rule #sizeWordStack ( WS ) => #sizeWordStack(WS, 0)
    rule #sizeWordStack ( .WordStack, SIZE ) => SIZE
    rule #sizeWordStack ( W : WS, SIZE )     => #sizeWordStack(WS, SIZE +Int 1)

    syntax Bool ::= Int "in" WordStack [function]
 // ---------------------------------------------
    rule W in .WordStack => false
    rule W in (W' : WS)  => (W ==K W') orElseBool (W in WS)
```

- `#padToWidth(N, WS)` makes sure that a `WordStack` is the correct size.

```
    syntax WordStack ::= #padToWidth ( Int , WordStack ) [function]
 // ---------------------------------------------------------------
    rule #padToWidth(N, WS) => WS                      requires notBool
↪#sizeWordStack(WS) <Int N [concrete]
    rule #padToWidth(N, WS) => #padToWidth(N, 0 : WS) requires #sizeWordStack(WS)␣
↪<Int N          [concrete]
```

- `WordStack2List` converts a term of sort `WordStack` to a term of sort `List`.

```
    syntax List ::= WordStack2List ( WordStack ) [function]
 // -------------------------------------------------------
    rule WordStack2List(.WordStack) => .List
    rule WordStack2List(W : WS) => ListItem(W) WordStack2List(WS)
```

13.2 Byte Arrays

The local memory of execution is a byte-array (instead of a word-array).

- `#asWord` will interperet a stack of bytes as a single word (with MSB first).
- `#asInteger` will interperet a stack of bytes as a single arbitrary-precision integer (with MSB first).
- `#asAccount` will interpret a stack of bytes as a single account id (with MSB first). Differs from `#asWord` only in that an empty stack represents the empty account, not account zero.
- `#asByteStack` will split a single word up into a `WordStack` where each word is a byte wide.

```
    syntax Int ::= #asWord ( WordStack ) [function, smtlib(asWord)]
 // --------------------------------------------------------------
    rule #asWord( .WordStack     ) => 0                               //␣
↪[concrete]
    rule #asWord( W : .WordStack ) => W                               //␣
↪[concrete]
    rule #asWord( W0 : W1 : WS   ) => #asWord(((W0 *Word 256) +Word W1) : WS)␣
↪[concrete]

    syntax Int ::= #asInteger ( WordStack ) [function]
 // --------------------------------------------------
    rule #asInteger( .WordStack     ) => 0
    rule #asInteger( W : .WordStack ) => W
    rule #asInteger( W0 : W1 : WS   ) => #asInteger(((W0 *Int 256) +Int W1) : WS)

    syntax Account ::= #asAccount ( WordStack ) [function]
 // ------------------------------------------------------
    rule #asAccount( .WordStack ) => .Account
    rule #asAccount( W : WS     ) => #asWord(W : WS)

    syntax WordStack ::= #asByteStack ( Int )              [function]
                       | #asByteStack ( Int , WordStack ) [function, klabel(
↪#asByteStackAux), smtlib(asByteStack)]
 // ---------------------------------------------------------------------------------
↪----------------------------
    rule #asByteStack( W ) => #asByteStack( W , .WordStack )                          ␣
↪              [concrete]
    rule #asByteStack( 0 , WS ) => WS                                                 ␣
↪           // [concrete]
    rule #asByteStack( W , WS ) => #asByteStack( W /Int 256 , W modInt 256 : WS )␣
↪requires W =/=K 0 [concrete]
```

13.3 Addresses

- `#addr` turns an Ethereum word into the corresponding Ethereum address (160 LSB).

```
    syntax Int ::= #addr ( Int ) [function]
 // ---------------------------------------
    rule #addr(W) => W %Word pow160
```

- `#newAddr` computes the address of a new account given the address and nonce of the creating account.
- `#sender` computes the sender of the transaction from its data and signature.

```
    syntax Int ::= #newAddr ( Int , Int ) [function]
 // ------------------------------------------------
    rule #newAddr(ACCT, NONCE) => #addr(#parseHexWord(Keccak256(#rlpEncodeLength(
↪#rlpEncodeBytes(ACCT, 20) +String #rlpEncodeWord(NONCE), 192))))

    syntax Int ::= #sender ( Int , Int , Int , Account , Int , String , Int ,␣
↪WordStack , WordStack ) [function]
                 | #sender ( String , Int , String , String )                      ␣
↪                 [function, klabel(#senderAux)]
                 | #sender ( String )                                              ␣
↪                 [function, klabel(#senderAux2)]
 // ------------------------------------------------------------------------------
↪-----------------------------------------------
    rule #sender(TN, TP, TG, TT, TV, DATA, TW, TR, TS)
      => #sender(#unparseByteStack(#parseHexBytes(Keccak256(#rlpEncodeLength(
↪#rlpEncodeWordStack(TN : TP : TG : .WordStack) +String #rlpEncodeAccount(TT)␣
↪+String #rlpEncodeWord(TV) +String #rlpEncodeString(DATA), 192)))), TW,
↪#unparseByteStack(TR), #unparseByteStack(TS))

    rule #sender(HT, TW, TR, TS) => #sender(ECDSARecover(HT, TW, TR, TS))

    rule #sender("")  => .Account
    rule #sender(STR) => #addr(#parseHexWord(Keccak256(STR))) requires STR =/=String "
↪"
```

- `#blockHeaderHash` computes the hash of a block header given all the block data.

```
    syntax Int ::= #blockHeaderHash( Int , Int , Int , Int , Int , Int , WordStack ,␣
↪Int , Int , Int , Int , Int , WordStack , Int , Int ) [function]
                 | #blockHeaderHash(String, String, String, String, String, String,␣
↪String, String, String, String, String, String, String, String, String) [function,␣
↪klabel(#blockHashHeaderStr)]
 // ------------------------------------------------------------------------------
↪------------------------------------------------------------------------------------
↪-------------------------
   rule #blockHeaderHash(HP, HO, HC, HR, HT, HE, HB, HD, HI, HL, HG, HS, HX, HM, HN)
         => #blockHeaderHash(#asWord(#parseByteStackRaw(HP)),
                             #asWord(#parseByteStackRaw(HO)),
                             #asWord(#parseByteStackRaw(HC)),
                             #asWord(#parseByteStackRaw(HR)),
                             #asWord(#parseByteStackRaw(HT)),
                             #asWord(#parseByteStackRaw(HE)),
                                     #parseByteStackRaw(HB) ,
                             #asWord(#parseByteStackRaw(HD)),
                             #asWord(#parseByteStackRaw(HI)),
                             #asWord(#parseByteStackRaw(HL)),
                             #asWord(#parseByteStackRaw(HG)),
                             #asWord(#parseByteStackRaw(HS)),
                                     #parseByteStackRaw(HX) ,
                             #asWord(#parseByteStackRaw(HM)),
                             #asWord(#parseByteStackRaw(HN)))

    rule #blockHeaderHash(HP, HO, HC, HR, HT, HE, HB, HD, HI, HL, HG, HS, HX, HM, HN)
         => #parseHexWord(Keccak256(#rlpEncodeLength(         #rlpEncodeBytes(HP, 32)
                                                      +String #rlpEncodeBytes(HO, 32)
                                                      +String #rlpEncodeBytes(HC, 20)
                                                      +String #rlpEncodeBytes(HR, 32)
```

(continues on next page)

(continued from previous page)

```
                                                            +String #rlpEncodeBytes(HT, 32)
                                                            +String #rlpEncodeBytes(HE, 32)
                                                            +String #rlpEncodeString(
↪#unparseByteStack(HB))
                                                            +String #rlpEncodeWordStack(HD␣
↪: HI : HL : HG : HS : .WordStack)
                                                            +String #rlpEncodeString(
↪#unparseByteStack(HX))
                                                            +String #rlpEncodeBytes(HM, 32)
                                                            +String #rlpEncodeBytes(HN, 8),
                                                          192)))
```

13.4 Word Map

Most of EVM data is held in finite maps. We are using the polymorphic `Map` sort for these word maps.

- `WM [ N := WS ]` assigns a contiguous chunk of $WM$ to $WS$ starting at position $W$.
- `#asMapWordStack` converts a `WordStack` to a `Map`.
- `#range(M, START, WIDTH)` reads off $WIDTH$ elements from $WM$ beginning at position $START$ (padding with zeros as needed).

```
    syntax Map ::= Map "[" Int ":=" WordStack "]" [function]
 // --------------------------------------------------------
    rule WM[ N := .WordStack ] => WM
    rule WM[ N := W : WS     ] => (WM[N <- W])[N +Int 1 := WS]

    syntax Map ::= #asMapWordStack ( WordStack ) [function]
 // -------------------------------------------------------
    rule #asMapWordStack(WS:WordStack) => .Map [ 0 := WS ]

    syntax WordStack ::= #range ( Map , Int , Int )            [function]
    syntax WordStack ::= #range ( Map , Int , Int , WordStack) [function, klabel(
↪#rangeAux)]
 // ------------------------------------------------------------------------------------
↪------
    rule #range(WM, START, WIDTH) => #range(WM, START +Int WIDTH -Int 1, WIDTH, .
↪WordStack)

    rule #range(WM,           END, 0,     WS) => WS
    rule #range(WM,           END, WIDTH, WS) => #range(WM, END -Int 1, WIDTH -Int 1,␣
↪0 : WS) requires (WIDTH >Int 0) andBool notBool END in_keys(WM)
    rule #range(END |-> W WM, END, WIDTH, WS) => #range(WM, END -Int 1, WIDTH -Int 1,␣
↪W : WS) requires (WIDTH >Int 0)
```

- `#removeZeros` removes any entries in a map with zero values.

```
    syntax Map ::= #removeZeros ( Map ) [function]
 // ----------------------------------------------
    rule #removeZeros( .Map )               => .Map
    rule #removeZeros( KEY |-> 0     REST ) => #removeZeros(REST)
    rule #removeZeros( KEY |-> VALUE REST ) => KEY |-> VALUE #removeZeros(REST)␣
↪requires VALUE =/=K 0
```

- `#lookup` looks up a key in a map and returns 0 if the key doesn't exist, otherwise returning its value.

```
    syntax Int ::= #lookup ( Map , Int ) [function]
 // ----------------------------------------------
    rule #lookup( (KEY |-> VAL) M, KEY ) => VAL
    rule #lookup(               M, KEY ) => 0 requires notBool KEY in_keys(M)
```

CHAPTER FOURTEEN

PARSING/UNPARSING

The EVM test-sets are represented in JSON format with hex-encoding of the data and programs. Here we provide some standard parser/unparser functions for that format.

14.1 Parsing

These parsers can interperet hex-encoded strings as `Int`s, `WordStack`s, and `Map`s.

- `#parseHexWord` interperets a string as a single hex-encoded `Word`.
- `#parseHexBytes` interperets a string as a hex-encoded stack of bytes.
- `#parseByteStack` interperets a string as a hex-encoded stack of bytes, but makes sure to remove the leading "0x".
- `#parseByteStackRaw` inteprets a string as a stack of bytes.
- `#parseWordStack` interperets a JSON list as a stack of `Word`.
- `#parseMap` interperets a JSON key/value object as a map from `Word` to `Word`.
- `#parseAddr` interperets a string as a 160 bit hex-endcoded address.

```
    syntax Int ::= #parseHexWord ( String ) [function]
                 | #parseWord    ( String ) [function]
 // --------------------------------------------------
    rule #parseHexWord("")    => 0
    rule #parseHexWord("0x") => 0
    rule #parseHexWord(S)     => String2Base(replaceAll(S, "0x", ""), 16) requires (S␣
↪=/=String "") andBool (S =/=String "0x")

    rule #parseWord("") => 0
    rule #parseWord(S)  => #parseHexWord(S) requires lengthString(S) >=Int 2 andBool␣
↪substrString(S, 0, 2) ==String "0x"
    rule #parseWord(S)  => String2Int(S) [owise]

    syntax WordStack ::= #parseHexBytes  ( String ) [function]
                       | #parseByteStack ( String ) [function]
                       | #parseByteStackRaw ( String ) [function]
 // --------------------------------------------------------------
    rule #parseByteStack(S) => #parseHexBytes(replaceAll(S, "0x", ""))
    rule #parseHexBytes("") => .WordStack
    rule #parseHexBytes(S)  => #parseHexWord(substrString(S, 0, 2)) :
↪#parseHexBytes(substrString(S, 2, lengthString(S))) requires lengthString(S) >=Int 2
```

(continues on next page)

(continued from previous page)

```
    rule #parseByteStackRaw(S) => ordChar(substrString(S, 0, 1)) :
↪#parseByteStackRaw(substrString(S, 1, lengthString(S))) requires lengthString(S)␣
↪>=Int 1
    rule #parseByteStackRaw("") => .WordStack

    syntax WordStack ::= #parseWordStack ( JSON ) [function]
 // -----------------------------------------------------------
    rule #parseWordStack( [ .JSONList ] )             => .WordStack
    rule #parseWordStack( [ (WORD:String) , REST ] ) => #parseHexWord(WORD) :
↪#parseWordStack( [ REST ] )

    syntax Map ::= #parseMap ( JSON ) [function]
 // ----------------------------------------------
    rule #parseMap( { .JSONList                     } ) => .Map
    rule #parseMap( { _   : (VALUE:String) , REST } ) => #parseMap({ REST })          ␣
↪                                   requires #parseHexWord(VALUE) ==K 0
    rule #parseMap( { KEY : (VALUE:String) , REST } ) => #parseMap({ REST }) [
↪#parseHexWord(KEY) <- #parseHexWord(VALUE) ] requires #parseHexWord(VALUE) =/=K 0

    syntax Int ::= #parseAddr ( String ) [function]
 // -------------------------------------------------
    rule #parseAddr(S) => #addr(#parseHexWord(S))
```

14.2 Unparsing

We need to interperet a `WordStack` as a `String` again so that we can call `Keccak256` on it from `KRYPTO`.

- `#unparseByteStack` turns a stack of bytes (as a `WordStack`) into a `String`.
- `#padByte` ensures that the `String` interperetation of a `Int` is wide enough.

```
    syntax String ::= #unparseByteStack ( WordStack )                [function,␣
↪klabel(unparseByteStack)]
                    | #unparseByteStack ( WordStack , StringBuffer ) [function,␣
↪klabel(#unparseByteStackAux)]
 // ------------------------------------------------------------------------------------
↪-----------------------
    rule #unparseByteStack ( WS ) => #unparseByteStack(WS, .StringBuffer)

    rule #unparseByteStack( .WordStack, BUFFER ) => StringBuffer2String(BUFFER)
    rule #unparseByteStack( W : WS, BUFFER )      => #unparseByteStack(WS, BUFFER␣
↪+String chrChar(W modInt (2 ^Int 8)))

    syntax String ::= #padByte( String ) [function]
 // -------------------------------------------------
    rule #padByte( S ) => S              requires lengthString(S) ==K 2
    rule #padByte( S ) => "0" +String S requires lengthString(S) ==K 1
```

CHAPTER

FIFTEEN

RECURSIVE LENGTH PREFIX (RLP)

RLP encoding is used extensively for executing the blocks of a transaction. For details about RLP encoding, see the YellowPaper Appendix B.

15.1 Encoding

- `#rlpEncodeWord` RLP encodes a single EVM word.
- `#rlpEncodeString` RLP encodes a single `String`.

```
    syntax String ::= #rlpEncodeWord ( Int )            [function]
                    | #rlpEncodeBytes ( Int , Int )      [function]
                    | #rlpEncodeWordStack ( WordStack ) [function]
                    | #rlpEncodeString ( String )        [function]
                    | #rlpEncodeAccount ( Account )      [function]
 // -------------------------------------------------------------
    rule #rlpEncodeWord(0) => "\x80"
    rule #rlpEncodeWord(WORD) => chrChar(WORD) requires WORD >Int 0 andBool WORD <Int␣
↪128
    rule #rlpEncodeWord(WORD) => #rlpEncodeLength(#unparseByteStack(
↪#asByteStack(WORD)), 128) requires WORD >=Int 128

    rule #rlpEncodeBytes(WORD, LEN) => #rlpEncodeString(#unparseByteStack(
↪#padToWidth(LEN, #asByteStack(WORD))))

    rule #rlpEncodeWordStack(.WordStack) => ""
    rule #rlpEncodeWordStack(W : WS)     => #rlpEncodeWord(W) +String
↪#rlpEncodeWordStack(WS)

    rule #rlpEncodeString(STR) => STR                           requires␣
↪lengthString(STR) ==Int 1 andBool ordChar(STR) <Int 128
    rule #rlpEncodeString(STR) => #rlpEncodeLength(STR, 128) [owise]

    rule #rlpEncodeAccount(.Account) => "\x80"
    rule #rlpEncodeAccount(ACCT)     => #rlpEncodeBytes(ACCT, 20) requires ACCT =/=K .
↪Account

    syntax String ::= #rlpEncodeLength ( String , Int )          [function]
                    | #rlpEncodeLength ( String , Int , String ) [function, klabel(
↪#rlpEncodeLengthAux)]
 // --------------------------------------------------------------------------------
↪------------------
    rule #rlpEncodeLength(STR, OFFSET) => chrChar(lengthString(STR) +Int OFFSET)␣
↪+String STR requires lengthString(STR) <Int 56
```

(continues on next page)

(continued from previous page)

```
    rule #rlpEncodeLength(STR, OFFSET) => #rlpEncodeLength(STR, OFFSET,
↪#unparseByteStack(#asByteStack(lengthString(STR)))) requires lengthString(STR)␣
↪>=Int 56
    rule #rlpEncodeLength(STR, OFFSET, BL) => chrChar(lengthString(BL) +Int OFFSET␣
↪+Int 55) +String BL +String STR
```

15.2 Decoding

- `#rlpDecode` RLP decodes a single `String` into a `JSON`.
- `#rlpDecodeList` RLP decodes a single `String` into a `JSONList`, interpereting the string as the RLP encoding of a list.

```
    syntax JSON ::= #rlpDecode(String)               [function]
                  | #rlpDecode(String, LengthPrefix)  [function, klabel(#rlpDecodeAux)]
 // ---------------------------------------------------------------------------------
    rule #rlpDecode(STR) => #rlpDecode(STR, #decodeLengthPrefix(STR, 0))
    rule #rlpDecode(STR, #str(LEN, POS))  => substrString(STR, POS, POS +Int LEN)
    rule #rlpDecode(STR, #list(LEN, POS)) => [#rlpDecodeList(STR, POS)]

    syntax JSONList ::= #rlpDecodeList(String, Int)                [function]
                      | #rlpDecodeList(String, Int, LengthPrefix) [function, klabel(
↪#rlpDecodeListAux)]
 // ---------------------------------------------------------------------------------
↪-----------------
    rule #rlpDecodeList(STR, POS) => #rlpDecodeList(STR, POS, #decodeLengthPrefix(STR,
↪ POS)) requires POS <Int lengthString(STR)
    rule #rlpDecodeList(STR, POS) => .JSONList [owise]
    rule #rlpDecodeList(STR, POS, _:LengthPrefixType(L, P)) =>
↪#rlpDecode(substrString(STR, POS, L +Int P)) , #rlpDecodeList(STR, L +Int P)

    syntax LengthPrefixType ::= "#str" | "#list"
    syntax LengthPrefix ::= LengthPrefixType "(" Int "," Int ")"
                          | #decodeLengthPrefix ( String , Int )                    ␣
↪           [function]
                          | #decodeLengthPrefix ( String , Int , Int )              ␣
↪           [function, klabel(#decodeLengthPrefixAux)]
                          | #decodeLengthPrefixLength ( LengthPrefixType , String ,␣
↪Int , Int ) [function]
                          | #decodeLengthPrefixLength ( LengthPrefixType , Int    ,␣
↪Int , Int ) [function, klabel(#decodeLengthPrefixLengthAux)]
 // ---------------------------------------------------------------------------------
↪-----------------------------------------------------------
    rule #decodeLengthPrefix(STR, START) => #decodeLengthPrefix(STR, START,␣
↪ordChar(substrString(STR, START, START +Int 1)))

    rule #decodeLengthPrefix(STR, START, B0) => #str(1, START)                      ␣
↪            requires B0 <Int 128
    rule #decodeLengthPrefix(STR, START, B0) => #str(B0 -Int 128, START +Int 1)     ␣
↪            requires B0 >=Int 128 andBool B0 <Int (128 +Int 56)
    rule #decodeLengthPrefix(STR, START, B0) => #decodeLengthPrefixLength(#str, STR,␣
↪START, B0)  requires B0 >=Int (128 +Int 56) andBool B0 <Int 192
    rule #decodeLengthPrefix(STR, START, B0) => #list(B0 -Int 192, START +Int 1)    ␣
↪            requires B0 >=Int 192 andBool B0 <Int 192 +Int 56
```

(continues on next page)

(continued from previous page)

```
    rule #decodeLengthPrefix(STR, START, B0) => #decodeLengthPrefixLength(#list, STR,␣
↪START, B0) [owise]

    rule #decodeLengthPrefixLength(#str,  STR, START, B0) =>
↪#decodeLengthPrefixLength(#str,  START, B0 -Int 128 -Int 56 +Int 1, #asWord(
↪#parseByteStackRaw(substrString(STR, START +Int 1, START +Int 1 +Int (B0 -Int 128␣
↪-Int 56 +Int 1)))))
    rule #decodeLengthPrefixLength(#list, STR, START, B0) =>
↪#decodeLengthPrefixLength(#list, START, B0 -Int 192 -Int 56 +Int 1, #asWord(
↪#parseByteStackRaw(substrString(STR, START +Int 1, START +Int 1 +Int (B0 -Int 192␣
↪-Int 56 +Int 1)))))
    rule #decodeLengthPrefixLength(TYPE, START, LL, L) => TYPE(L, START +Int 1 +Int␣
↪LL)
endmodule
```

CHAPTER

SIXTEEN

ANALYSIS TOOLS

Here, we define analysis tools specific to EVM. These tools are defined as extensions of the semantics, utilizing the underlying machinery to do execution. One benefit of K is that we do not have to re-specify properties about the operational behavior in our analysis tools; instead we can take the operational behavior directly.

```
requires "evm.k"

module EVM-ANALYSIS
    imports EVM
```

16.1 Gas Analysis

Gas analysis tools will help in determining how much gas to call a contract with given a specific input. This can be used to ensure that computation will finish without throwing an exception and rolling back state. Here we provide a simplistic gas analysis tool (which just returns an approximation of the gas used for each basic block). This tool should be extended to take advantage of the symbolic execution engine so that we can provide proper bounds on the gas used.

- The mode `GASANALYZE` performs gas analysis of the program instead of executing normally.

```
    syntax Mode ::= "GASANALYZE" [klabel(GASANALYZE)]
```

We'll need to make summaries of the state which collect information about how much gas has been used.

- `#beginSummary` appends a new (unfinished) summary entry in the `analysis` cell under the key `"gasAnalyze"`.
- `#endSummary` looks for an unfinished summary entry by the key `"gasAnalyze"` and performs the substraction necessary to state how much gas has been used since the corresponding `#beginSummary`.

```
    syntax Summary ::= "{" Int "|" Int "|" Int "}"
                     | "{" Int "==>" Int "|" Int "|" Int "}"
 // -------------------------------------------------------

    syntax InternalOp ::= "#beginSummary"
 // -------------------------------------
    rule <k> #beginSummary => . ... </k> <pc> PCOUNT </pc> <gas> GAVAIL </gas>
↪<memoryUsed> MEMUSED </memoryUsed>
         <analysis> ... "blocks" |-> ((.List => ListItem({ PCOUNT | GAVAIL | MEMUSED }
↪)) REST) ... </analysis>

    syntax KItem ::= "#endSummary"
 // ------------------------------
```

(continues on next page)

(continued from previous page)

```
    rule <statusCode> EVMC_SUCCESS </statusCode> <k> (#halt => .) ~> #endSummary ...
↪</k>
    rule <k> #endSummary => . ... </k> <pc> PCOUNT </pc> <gas> GAVAIL </gas>
↪<memoryUsed> MEMUSED </memoryUsed>
         <analysis> ... "blocks" |-> (ListItem({ PCOUNT1 | GAVAIL1 | MEMUSED1 } => {␣
↪PCOUNT1 ==> PCOUNT | GAVAIL1 -Int GAVAIL | MEMUSED -Int MEMUSED1 }) REST) ... </
↪analysis>
```

- In GASANALYZE mode, summaries of the state are taken at each #gasBreaks opcode, otherwise execution is as in NORMAL.

```
    rule <mode> GASANALYZE </mode>
         <k> #next => #setMode NORMAL ~> #execTo #gasBreaks ~> #setMode GASANALYZE ...
↪ </k>
         <pc> PCOUNT </pc>
         <program> ... PCOUNT |-> OP ... </program>
      requires notBool (OP in #gasBreaks)

    rule <mode> GASANALYZE </mode>
         <k> #next => #endSummary ~> #setPC (PCOUNT +Int 1) ~> #setGas 1000000000 ~>
↪#beginSummary ~> #next ... </k>
         <pc> PCOUNT </pc>
         <program> ... PCOUNT |-> OP ... </program>
      requires OP in #gasBreaks

    syntax Set ::= "#gasBreaks" [function]
 // --------------------------------------
    rule #gasBreaks => (SetItem(JUMP) SetItem(JUMPI) SetItem(JUMPDEST))

    syntax InternalOp ::= "#setPC"  Int
                        | "#setGas" Int
 // -----------------------------------
    rule <k> #setPC PCOUNT  => . ... </k> <pc> _ => PCOUNT </pc>
    rule <k> #setGas GAVAIL => . ... </k> <gas> _ => GAVAIL </gas>
```

- #gasAnalyze analyzes the gas of a chunk of code by setting up the analysis state appropriately and then setting the mode to GASANALYZE.

```
    syntax KItem ::= "#gasAnalyze"
 // ------------------------------
    rule <k> #gasAnalyze => #setGas 1000000000 ~> #beginSummary ~> #setMode␣
↪GASANALYZE ~> #execute ~> #endSummary ... </k>
         <pc> _ => 0 </pc>
         <gas> _ => 1000000000 </gas>
         <analysis> _ => ("blocks" |-> .List) </analysis>
endmodule
```

CHAPTER

SEVENTEEN

CRYPTOGRAPHIC PRIMITIVES

Here we implement the various cryptographic primitives needed for KEVM.

```
module KRYPTO
    imports STRING-SYNTAX
    imports INT-SYNTAX
    imports LIST
```

- `Keccak256` takes a string and returns a 64-character hex-encoded string of the 32-byte keccak256 hash of the string.
- `Sha256` takes a String and returns a 64-character hex-encoded string of the 32-byte SHA2-256 hash of the string.
- `RipEmd160` takes a String and returns a 40-character hex-encoded string of the 20-byte RIPEMD160 hash of the string.
- `ECDSARecover` takes a 32-character byte string of a message, v, r, s of the signed message and returns the 64-character public key used to sign the message. See this StackOverflow post for some information about v, r, and s.

```
    syntax String ::= Keccak256 ( String )                            [function,␣
↪hook(KRYPTO.keccak256)]
                    | ECDSARecover ( String , Int , String , String )  [function,␣
↪hook(KRYPTO.ecdsaRecover)]
                    | Sha256 ( String )                               [function,␣
↪hook(KRYPTO.sha256)]
                    | RipEmd160 ( String )                            [function,␣
↪hook(KRYPTO.ripemd160)]
 // ----------------------------------------------------------------------------------
↪------------------
```

The BN128 elliptic curve is defined over 2-dimensional points over the fields of zero- and first-degree polynomials modulo a large prime. (x, y) is a point on G1, whereas (x1 x x2, y1 x y2) is a point on G2, in which x1 and y1 are zero-degree coefficients and x2 and y2 are first-degree coefficients. In each case, (0, 0) is used to represent the point at infinity.

- `BN128Add` adds two points in G1 together,
- `BN128Mul` multiplies a point in G1 by a scalar.
- `BN128AtePairing` accepts a list of points in G1 and a list of points in G2 and returns whether the sum of the product of the discrete logarithm of the G1 points multiplied by the discrete logarithm of the G2 points is equal to zero.
- `isValidPoint` takes a point in either G1 or G2 and validates that it actually falls on the respective elliptic curve.

```
    syntax G1Point ::= "(" Int "," Int ")"
    syntax G2Point ::= "(" Int "x" Int "," Int "x" Int ")"
    syntax G1Point ::= BN128Add(G1Point, G1Point) [function, hook(KRYPTO.bn128add)]
                     | BN128Mul(G1Point, Int)      [function, hook(KRYPTO.bn128mul)]
 // --------------------------------------------------------------------------------

    syntax Bool ::= BN128AtePairing(List, List) [function, hook(KRYPTO.bn128ate)]
 // ------------------------------------------------------------------------------

    syntax Bool ::= isValidPoint(G1Point) [function, hook(KRYPTO.bn128valid)]
                  | isValidPoint(G2Point) [function, klabel(isValidG2Point),␣
↪hook(KRYPTO.bn128g2valid)]
 // -----------------------------------------------------------------------------------
↪------------------
endmodule
```

CHAPTER

EIGHTEEN

EVM DESIGN ISSUES

The EVM was the first successful general-purpose distributed programmable blockchain platform, but that doesn't make it without fault. There are several issues with both the specification of the EVM (in the Yellow Paper), and with the general design of the EVM. Most of these issues are written from the perspective of someone trying to do formal verification of the EVM.

18.1 Issues with description of EVM

These can be ambiguities/confusing wording in the Yellow Paper.

- In section 9.4.2, exceptions are described as if they are all catchable before an opcode is executed. While you may be able to implement EVM in this way, it's not clear that it's best (you have to duplicate computation), and we also are pretty sure no implementation even does this (including the C++ one). Instead they throw exceptions when they happen and roll-back the state (which is what you would expect to happen). Our original implementation tried to do it the way the Yellow Paper described, and it made everything harder/slower.
- Again in section 9.4.2, it specifies that "these are the only ways exceptions can happen when executing". This doesn't help with building implementations, because there is at least one other case that isn't described. For example, if the memory is overflown, then the existing semantics doesn't do anything. Should it throw an exception?
- Some operators which access data of other accounts don't specify explicitly what to do if the other account doesn't exist. `EXTCODESIZE` and `EXTCODECOPY` examples, though strangely enough `BALANCE` does specify what to do. We think the community has reached agreement on this though, "non-existing accounts are empty accounts" or something along those lines.
- What about contracts that have "junk bytes" in them? We've seen a contract with "junk bytes", and use cases of contracts with junk bytes do exist. For example, if you want to use some large chunk of data to be used in your contract but don't want to perform a sequences of `PUSH`, `CODECOPY` can be used to move the junk bytes into memory.
- The description in Appendix H of the `CALLCODE` instruction describes it as like `CALL` except for the fourth argument to the Theta function. However, it does not mention that this change from `Mu_s[1]` to `I_a` also applies to the specification of `C_NEW`.
- The description in Appendix H of the `DELEGATECALL` instruction describes the gas provided to the caller as equal to `Mu_s[0]`. However, this is clearly not the correct behavior, since `Mu_s[0]` is a user-provided value, and the user could set it equal to 2^256 - 1, leading to the user having an infinite amount of gas. It's clear from the test suite that the intended behavior is to use `Ccallgas` but with the value for the value transfer equal to 0. It also describes the exceptional condition of not enough balance in terms of `I_v`, but in fact no value transfer occurs so this condition should never occur.

18.2 Issues with design of EVM

More broadly, many features of the EVM seem to be poorly designed. These can be issues from simple “why did they do it that way?” to “this makes doing formal reasoning about EVM harder”.

- Precompiled contracts: Why are there 4 precompiled contracts? Calls into address 1 - 4 result in a “precompiled” contract being called (most of them some sort of cryptographic function). There are plenty of opcodes free, we should just have those precompiled contracts be accessed through primitives (like how `SHA3` is done). Another (albeit unlikely problem) is that of address-space collisions.
- The byte-aligned local memory makes reasoning about EVM programs much more difficult. Say, for instance, that you write two Words (256-bit) contiguously to local memory (which takes up addresses 0 - 64), then shift between them and read a word (say between addresses 16 - 48). If one of those words was symbolic, the resulting symbolic word is a mess of an expression involving the original words. Of course, in theory this is possible to reason about, but effectively this allows taking one symbolic value and turning it into 32 symbolic values. This makes symbolic execution much slower/more painful. Note also that attempting to use bit-vectors, where you have one symbolic boolean variable per bit, is currently infeasible with the existing SMT solvers like Z3; while it works with 32-bit words in some program verifiers, it is disarmingly slow with 64-bit words and we failed to prove anything with 256-bit words.
- Program representation is important in EVM (that is, you must be able to represent a program as a byte-array of opcodes). When doing program analysis/abstract verification, you ideally would be allowed to make transformations on the program representation (e.g., convert it to a control-flow graph) without having to maintain a translation back. Currently in EVM, the `*CODECOPY` opcodes allow regarding program pieces as data, meaning that a translation back must always be maintained, because using `CREATE` with `DELEGATECALL` allows executing arbitrary code. For this reason, we had to build a parser/unparser and an assembler/dissasembler into our semantics. Putting a symbolic value through the process of disassembling -> unparsing loses a lot of semantic information about the original value. While self-modifying code is nice and powerful in principle, we are not aware of any programming languages for the blockchain that encourage or even allow that.
- In section H.2, the Yellow Paper states “All arithmetic is modulo 2^256 unless otherwise noted.” Reasoning “modulo” is very complex with the current SMT provers and it was indeed a, if not the most major difficulty in our EVM verification efforts. Additionally, the programs (smart contracts) we verified turned out to be wrong, in the sense that they showed unintended behavior, in the presence of arithmetic overflows anyway. That is, arithmetic overflows were not expected to happen by the developers, so adding code to deal with the “modulo 2^256” behavior in case of arithmetic overflow was not even considered. In such situations, it would be a lot better to simply throw an exception when arithmetic overflow occurs, thus indicating that something bad happened, than to default to “modulo 2^256” and ending up with a program computing wrong values. We conjecture that words of 256 bits should be long enough for the current smart contract needs to afford to abruptly terminate computations when the limit is reached.

18.3 Recommendations for the Future

In addition to the above mentioned issues, there are several things that could be improved about EVM in general as a distributed computation language. Here we mention some.

18.3.1 Deterministic vs. Nondeterministic (and Proof of Work and Scalability)

Because EVM is deterministic, it takes as long to verify a computation as it takes to run a computation. In both cases, the entire program must be executed; there is no choice about what the next step to take is.

In a nondeterministic language, execution is finding one execution path among many which “solves” the program. For example, any logical language where there are several possible next inference steps is nondeterministic (eg. Prolog, Maude, K, Coq). However, once a solution is found, presenting it is telling which choices were made at each nondeterministic step; verifying it is following that same sequence of steps. If at each step there are a choice from `M` inference rules, and it takes `n` steps to reach a solution, then the speedup in verifying is `M^n`.

One of the goals in a consensus-driven distributed store is scalability, which means as more resources are added to the network the network gets stronger. Using a deterministic language means that we lose at least one dimension of this scalability; everyone verifying the state of the world must do as much work as it took to compute the state of the world. Even many functional languages, by having evaluation strategies settled ahead of time, are deterministic (though they may have elegant ways of encoding nondeterministic systems).

On the other hand, what secures many of these blockchain-consensus systems is proof of work. Proof of work is the ultimate non-deterministic programming language; the programs are the blocks (before adding the nonce), and the solutions are the nonce added to the blocks so that it hashes low enough. When using a nonce of size `2^N`, there are exactly `M = 2^N` next "inference steps", and they all must be searched uniformly to find a solution. If instead the underlying programming language had some nondeterminism, some of the proof of work could be done *just by executing the transactions going into the block*. Perhaps the two can be used to augment each other, allowing for some of the proof of work to be provided via finding a solution to the program and the rest via hashing.

If such a system were implemented, it may be important to incentivize miners to supply solutions to programs/proofs on the blockchain. Perhaps a system where the time between when a specification/theorem is submitted to the blockchain and when it is solved determines the reward for the computation could be used. Natural incentive to place proofs of theorems on the blockchain would be provided in the form of the reward; this means it's against the miners interests to ignore transactions. The hard part is incentivizing placing theorems on the blockchain early (as it may be advantageous to hoard theorems so that you can submit solutions early to collect the reward).

18.3.2 Termination

The gas mechanic in EVM is designed to ensure that every program terminates so that users can't DOS the miners by submitting infinite computations. However, there is no such guarantee that the proof of work computation done by miners terminates; there may be no combination of ordering of transactions and a nonce that yields a solution (though this is incredibly improbable). Instead, we can leave it up to the miner to decide if pursuing a computation is worth the time lost in the pursuit. Indeed, this directly increases the amount of work possible behind a proof of work, as much more useless work has been added to the system (via computations that don't terminate).

In many sufficiently powerful nondeterministic languages, there will be plenty of execution search paths which do not terminate. However, automated provers (execution engines) for these languages don't throw up their hands, instead they design better search tactics for the language. It's not clear that leaving the burden of which transactions to attempt to the miner is entirely bad, especially when coupled with a system which rewards more for longer-standing transactions.

The problem with this, it turns out, is not that users may DOS miners, but that miners may DOS other miners (by presenting blocks that they purport terminate).

18.3.3 Language Independence

Language independence is difficult to achieve in a distributed system because everyone must agree on how programs are to be executed. Two approaches are the *language-building language* approach and the *consensus-based* approach.

In the language-building language approach, the underlying language of the blockchain is a language-building language. Thus, contracts are free to introduce new languages simply as specifications (programs) in the underlying language, and other contracts may use those languages by referring to the language definition contract. As a very simple example, if the underlying language was K, then you could submit a contract that is just a K definition giving semantics to the language you want to use in the future. Along this line, we should use a logical framework as the underlying language. Logical frameworks exhibit both non-determinism and language independence, making two improvements to EVM at the same time.

The consensus-based approach is more flexible in the interpretation of "correct" executions of programs. Essentially, everyone would vote on which execution is correct by rejecting ill-formed blocks (ill-formed here includes blocks which do not report a correct execution). This lets the definition of the underlying languages evolve out of band; major changes to the semantics would essentially require widespread network agreement or a fork. Indeed, the only thing that should be stored on the blockchain would be a hash of the program.

These two techniques could perhaps be combined.

CHAPTER

NINETEEN

INDICES AND TABLES

- genindex
- modindex
- search