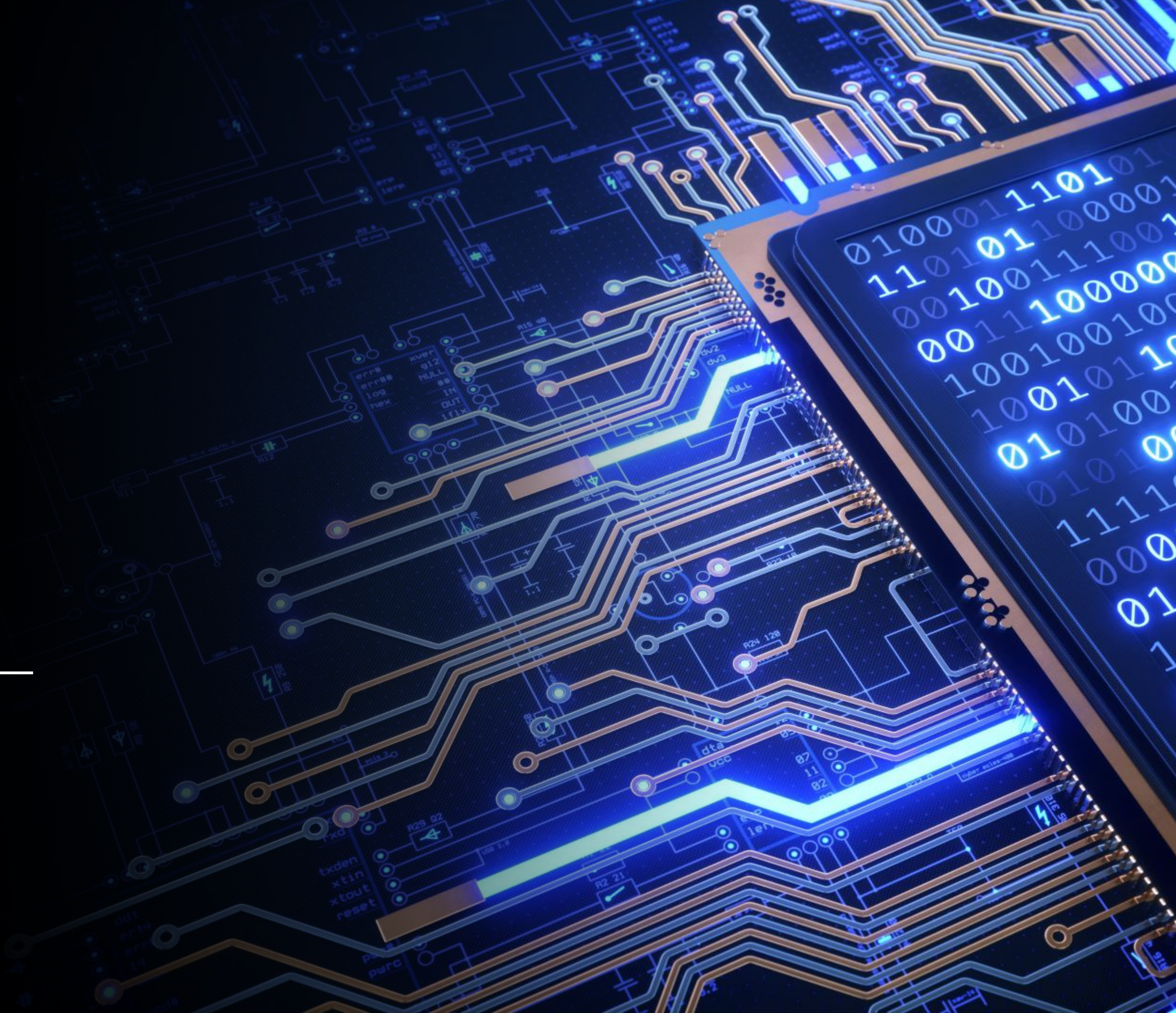
KEVM
K Ethereum Virtual Machine



Agenda

Agenda

- **Motivation**
- **Background**
  - BNF
  - K Framework
  - EVM
- **KEVM**
  - K Semantics of EVM
  - Semantic Evaluation
  - Derived Analysis Tools
  - Feature Comparison Overview

Motivation

Motivation

Example of C Program

- What should the following program evaluate to?

```
int main(void) {
    int x = 0;
    return (x = 1) + (x = 2);
}
```

Motivation

Example of C Program

- What should the following program evaluate to?

```
int main(void) {
    int x = 0;
    return (x = 1) + (x = 2);
}
```

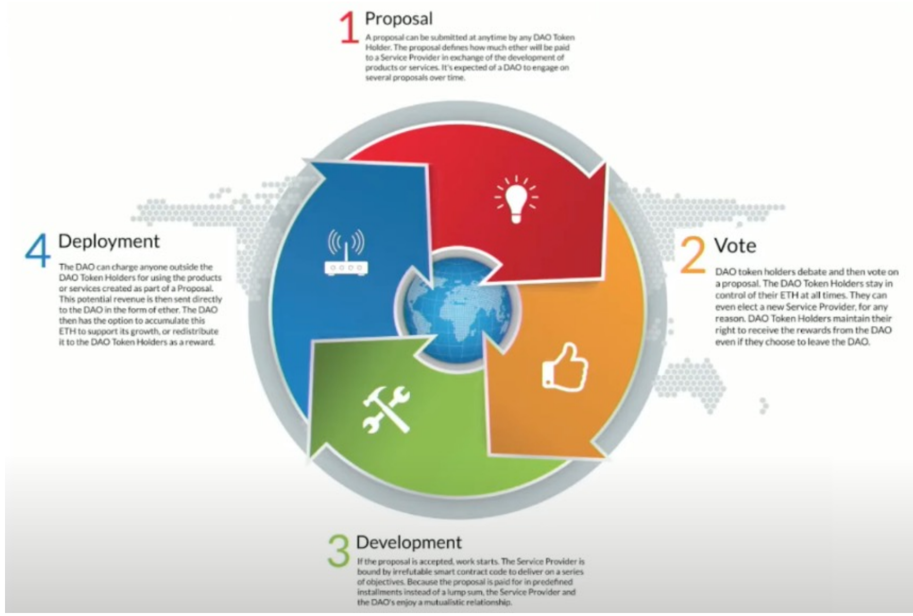
  - According to the C “standard”, It is undefined
  - GCC3, ICC, Clang returns 3
  - GCC4, MSVC return 4
- Formal program verifiers actually “proved” it retuns 4

  How can we trust any program verification claim if advanced formal analysis tools can prove correct about what is obviously wrong?

DAO Attack – Reminder

Motivation

- The DAO was a digital decentralized autonomous organization
- The DAO had an objective to provide a new decentralized business model for organizing both commercial and non-profit enterprises
- Effectively it was a Smart Contract that held over $150 million.
- The attackers used vulnerability found in the contract and was able to steal over $50 million worth of Ether.
- This attack caused the entire Ethereum team to get involved in order to save its reputation.
- In order to return the stolen funds hard fork has been done.



1 Proposal

A proposal can be submitted at anytime by any DAO Token Holder. The proposal defines how much ether will be paid to a Service Provider in exchange of the development of products or services. It's expected of a DAO to engage on several proposals over time.

2 Vote

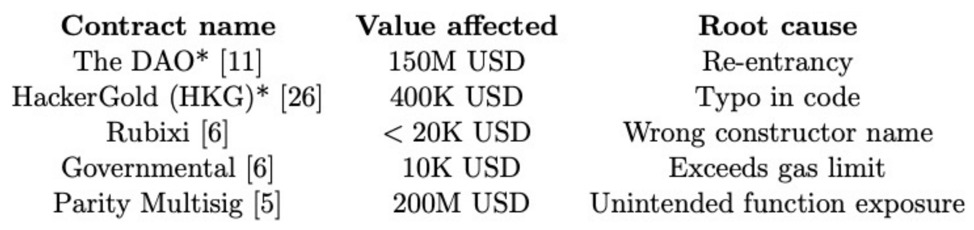
DAO token holders debate and then vote on a proposal. The DAO Token Holders stay in control of their ETH at all times. They can even elect a new Service Provider, for any reason. DAO Token Holders maintain their right to receive the rewards from the DAO even if they choose to leave the DAO.

3 Development

If the proposal is accepted, work starts. The Service Provider is bound by irrefutable smart contract code to deliver on a series of objectives. Because the proposal is paid for in predefined installments instead of a lump sum, the Service Provider and the DAO's enjoy a mutualistic relationship.

4 Deployment

The DAO can charge anyone outside the DAO Token Holders for using the products or services created as part of a Proposal. This potential revenue is then sent directly to the DAO in the form of ether. The DAO then has the option to accumulate this ETH to support its growth, or redistribute it to the DAO Token Holders as a reward.



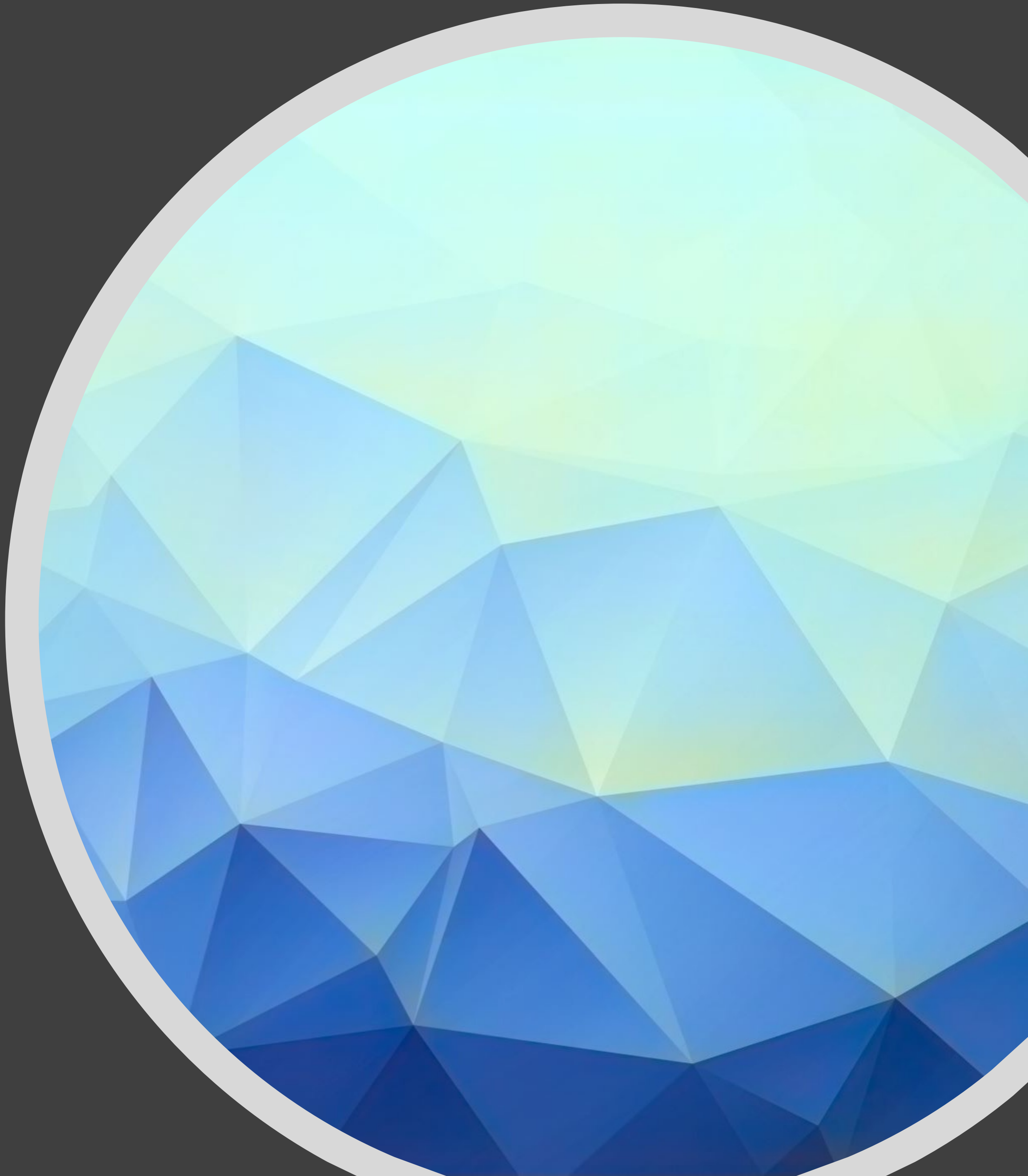
Contract name	Value affected	Root cause
The DAO* [11]	150M USD	Re-entrancy
HackerGold (HKG)* [26]	400K USD	Typo in code
Rubixi [6]	< 20K USD	Wrong constructor name
Governmental [6]	10K USD	Exceeds gas limit
Parity Multisig [5]	200M USD	Unintended function exposure

Background

BNF – Backus-Naur Form

K Framework

Ethereum Virtual Machine



BNF

Context free grammar. Idealy suited for describing the syntax of programming languages.

Basic Definitions:

- ::= *reads as* “is defined as”
- | *reads as* “or”
- <> Used to surround category names
- ~= *reads as* “followed by”
- => *reads as* “rewrites to”
- |-> *reads as* “look up”

BNF

Example

<digit> → 0|1|2|3|4|5|6|7|8|9

<natural> → <digit> | <digit><natural>

Example:

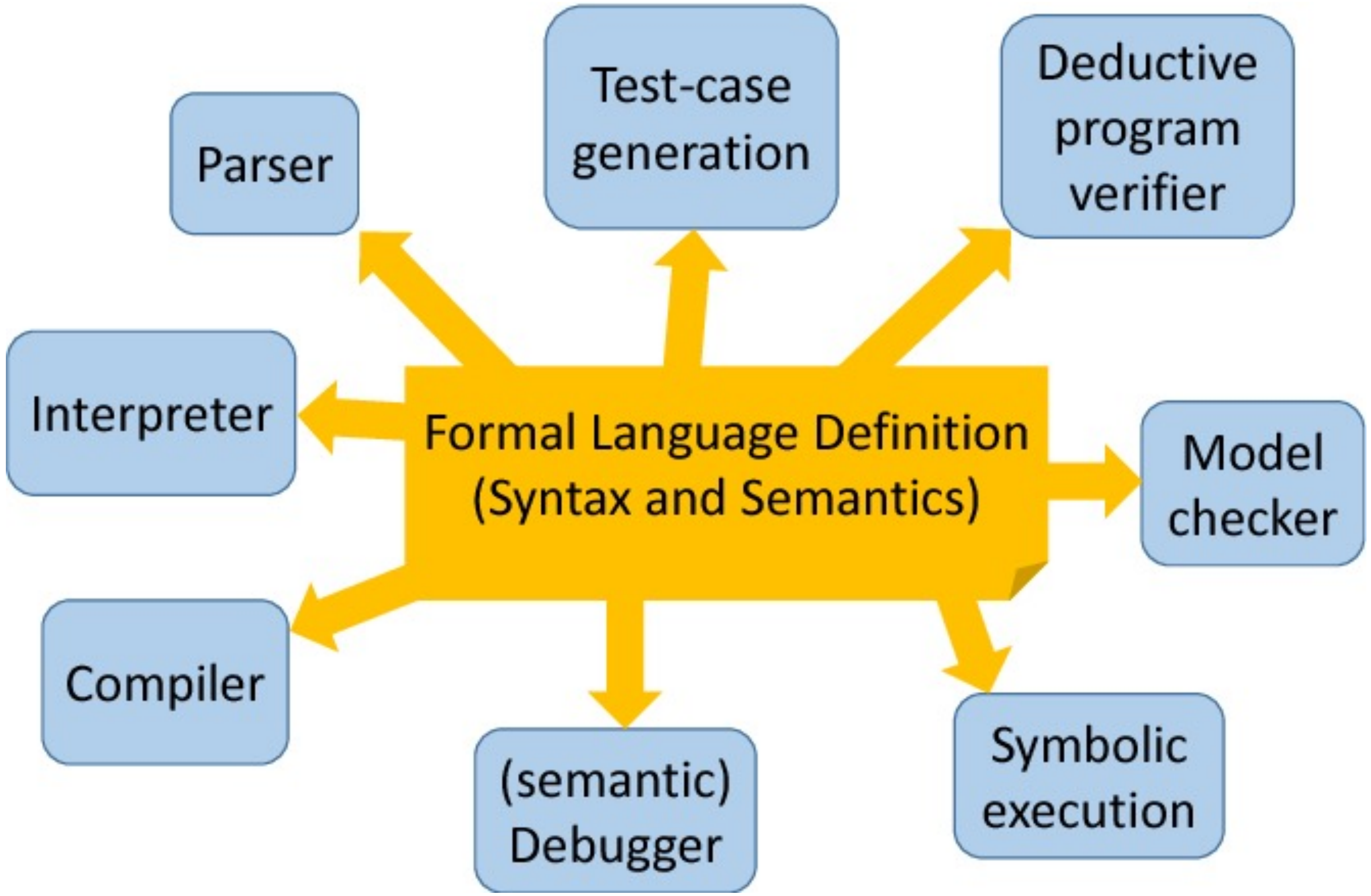
```
<natural> => <digit><natural>
          => 9<natural>
          => 9<digit>
          => 95
```

K Framework

Consider some programming language, L:

- Formal Semantics of L?
  Typically skipped – considered expensive & useless
- Implementations for L
  Based on adhoc understanding of what L is
- Model Checkers for L
  Based on some adhoc encodings / models of L
- Program Verifieirs for L.
  Based on some other adhoc encodings / models of L
- Etc.

K Framework


Parser
Test-case generation
Deductive program verifier
Interpreter
Formal Language Definition (Syntax and Semantics)
Model checker
Compiler
(semantic) Debugger
Symbolic execution

K Framework

What is K

- K is language for building Programming Languages
- When building PL using K,
  one can derive the aforementioned tools from it.
- PL Expert gives the formal language definitions (Syntax & Semantic) and get K Framework derive the tools from it.

K Backend

- Matching Logic - unifying foundational logic for programming languages, specification, verification.
- Reachabillitiy Logic - logic for reasoning symbolically about potentially infinite transition systems

K Framework

A Formal Semantics Manifesto

- Programming languages must have formal semantics
  - Analysis / Verification tools should build on them (o.w. they are adhoc and likely wrong)
- Informal Manuals are not sufficient
  - Manuals typically have a formal syntax of the language (in an appendix)
- In other words – separate development of PL software into two task:
  1. The Programming Language
  2. The Tooling (Parser, Interpeter, Debugger, Compiiler, Model Checker, Program Verifier)

Syntax

```
syntax Exp ::= Int | Id | “(“ Exp “)” [bracket]
               | Exp “*” Exp
               > Exp “+” Exp // looser binding

syntax Stmt ::= Id “:=“ Exp
               | Stmt “;” Stmt
               | “return” Exp
```

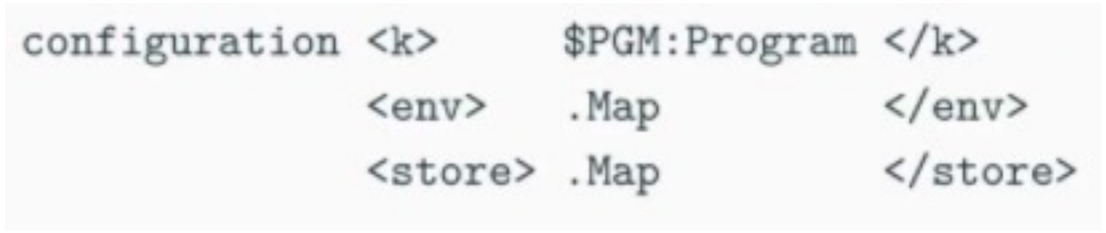
This would allow correctly parsing programs like:

```
a := 3 * 2;
b := 2 * a + 5;
return b
```

K Framework

Configuration

Tell K about the structure of your execution state. For example, a simple imperative language might have:



```
configuration <k>     $PGM:Program </k>
              <env>   .Map         </env>
              <store> .Map         </store>
```

- <k> will contain the initial parsed program.
- <env> contains bindings of variable names to store locations
- <store> contains bindings od store location to integers

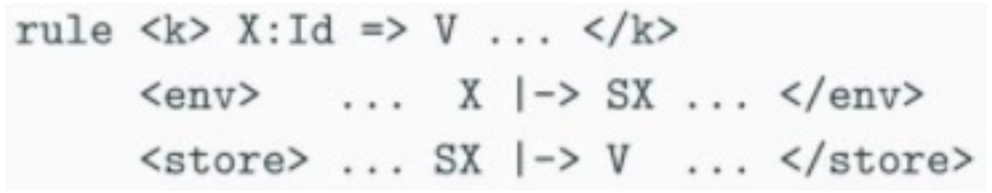
K Framework

Transition Rules

Rules define how the system transitions from one state to the next

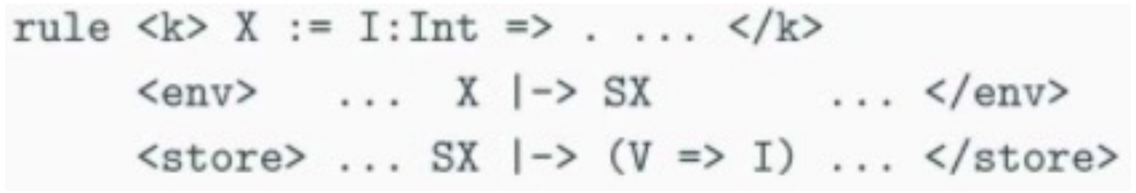
Using the above grammar and configuration:

Variable lookup



```
rule <k> X:Id => V ... </k>
     <env>   ...  X |-> SX ... </env>
     <store> ... SX |-> V  ... </store>
```

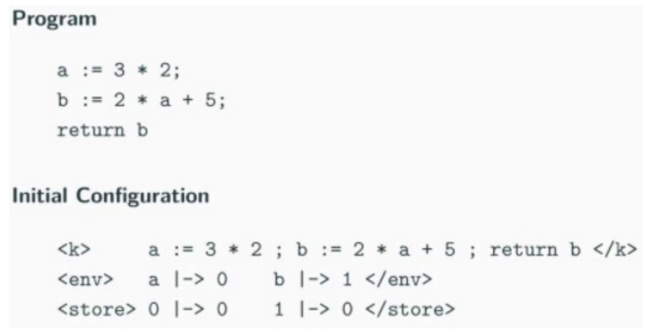
Variable assignment



```
rule <k> X := I:Int => . ... </k>
     <env>   ...  X |-> SX        ... </env>
     <store> ... SX |-> (V => I) ... </store>
```

K Framework

Example Execution

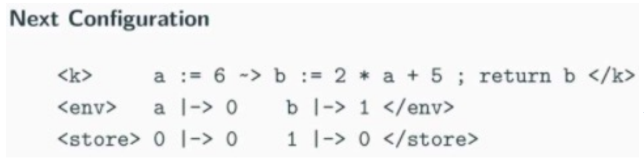


Program

```
a := 3 * 2;
b := 2 * a + 5;
return b
```

Initial Configuration

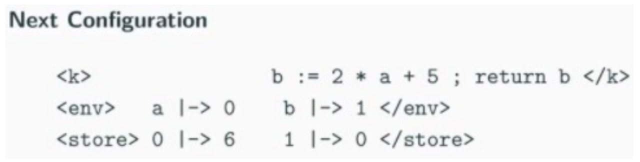
```
<k>     a := 3 * 2 ; b := 2 * a + 5 ; return b </k>
<env>   a |-> 0     b |-> 1 </env>
<store> 0 |-> 0     1 |-> 0 </store>
```



Next Configuration

```
<k>      a := 6 ~> b := 2 * a + 5 ; return b </k>
<env>    a |-> 0     b |-> 1 </env>
<store>  0 |-> 0     1 |-> 0 </store>
```

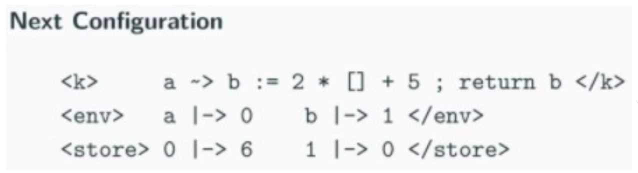
Variable Assignment



Next Configuration

```
<k>                 b := 2 * a + 5 ; return b </k>
<env>   a |-> 0     b |-> 1 </env>
<store> 0 |-> 6     1 |-> 0 </store>
```

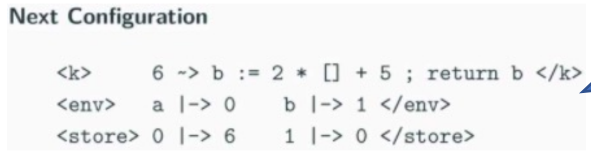
Variable Assignment



Next Configuration

```
<k>      a ~> b := 2 * [] + 5 ; return b </k>
<env>    a |-> 0     b |-> 1 </env>
<store>  0 |-> 6     1 |-> 0 </store>
```

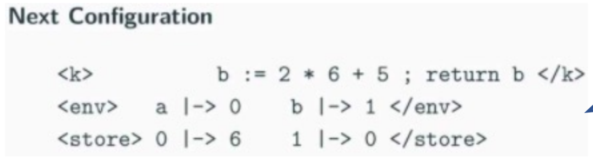
Variable Lookup



Next Configuration

```
<k>      6 ~> b := 2 * [] + 5 ; return b </k>
<env>    a |-> 0     b |-> 1 </env>
<store>  0 |-> 6     1 |-> 0 </store>
```

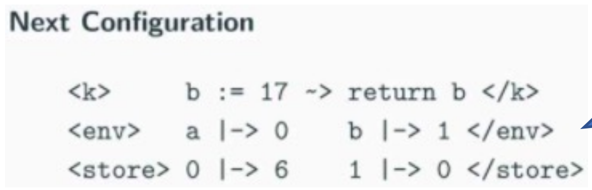
Variable Lookup



Next Configuration

```
<k>                b := 2 * 6 + 5 ; return b </k>
<env>    a |-> 0     b |-> 1 </env>
<store>  0 |-> 6     1 |-> 0 </store>
```

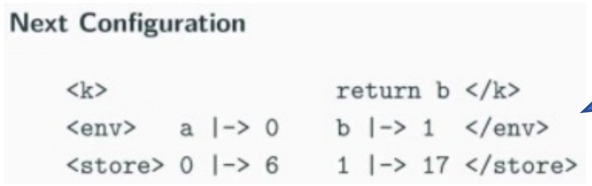
Variable Lookup



Next Configuration

```
<k>      b := 17 ~> return b </k>
<env>    a |-> 0     b |-> 1 </env>
<store>  0 |-> 6     1 |-> 0 </store>
```

Variable Assignment

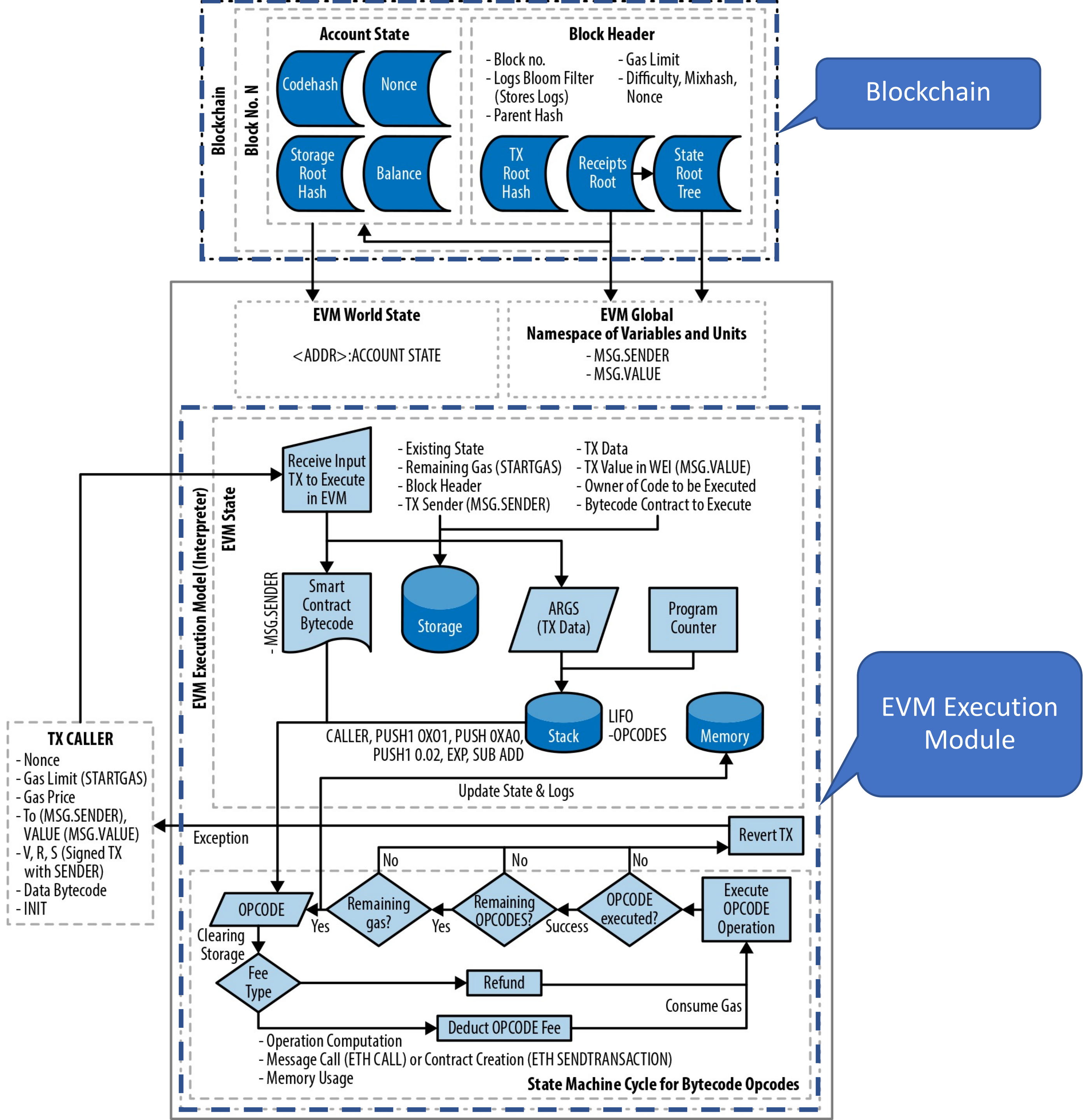


Next Configuration

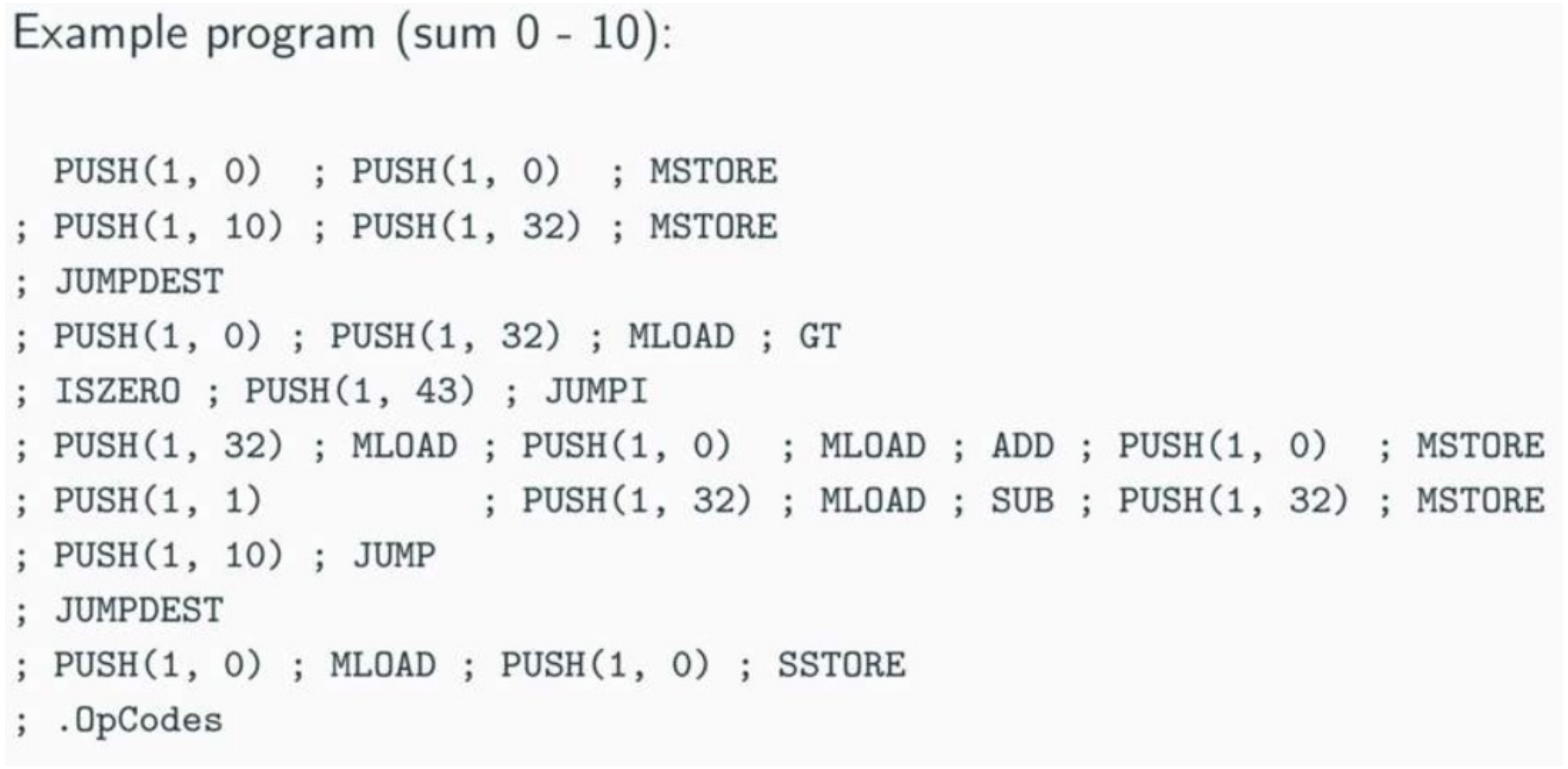
```
<k>                  return b </k>
<env>    a |-> 0     b |-> 1  </env>
<store>  0 |-> 6     1 |-> 17 </store>
```

Variable Assignment

Ethereum Virtual Machine



Ethereum Virtual Machine



Example program (sum 0 - 10):

```
  PUSH(1, 0)  ; PUSH(1, 0)  ; MSTORE
; PUSH(1, 10) ; PUSH(1, 32) ; MSTORE
; JUMPDEST
; PUSH(1, 0) ; PUSH(1, 32) ; MLOAD ; GT
; ISZERO ; PUSH(1, 43) ; JUMPI
; PUSH(1, 32) ; MLOAD ; PUSH(1, 0)  ; MLOAD ; ADD ; PUSH(1, 0)  ; MSTORE
; PUSH(1, 1)          ; PUSH(1, 32) ; MLOAD ; SUB ; PUSH(1, 32) ; MSTORE
; PUSH(1, 10) ; JUMP
; JUMPDEST
; PUSH(1, 0) ; MLOAD ; PUSH(1, 0) ; SSTORE
; .OpCodes
```

Ethereum
Virtual
Machine

Enforcing Termination (Gas) – Quick Reminder

User submitted transactions could have infinite loops (DoS Attacks)

- Solution:
  - Each opcode costs some gas, paid for in Ether.
  - Spent gas aawarded to miner, remaining refunded to account.

Notes

- It’s important to charge according to the actual used compute resources
- Tuning gas costs is an ongoing challenge:
  new hardware → new gas model

Ethereum
Virtual
Machine

Intercontract Executions & EVM ABI

- Contracts can call other contracts (called re-entrancy)
- Payload to other contract is a raw string of bytes called ***CallData***
- External to the EVM, the Ethereum ABI has been developed, which specifies:
    - Calling conventions (how to interpret ***CallData*** correctly)
    - Some high-level types (and their mapping to EVM words)

KEVM

K Semantic of EVM

Semantic Evaluation

Derived Analysis Tools

Feature Comparison Overview

K Semantics of EVM

K Semantics
of
EVM

Instantiating K with an EVM semantics requires:

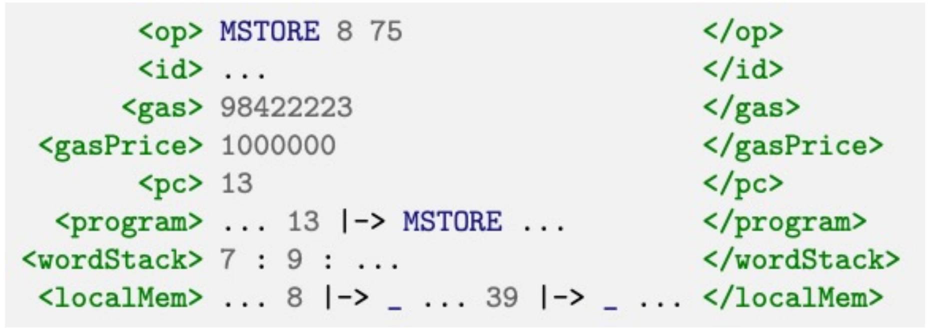
- modeling transaction execution state and network evolution using the ***configuration***
- detailing changes in transaction execution and network evolution using transition ***rules***

K Semantics
of
EVM

EVM Execution State (Configuration)

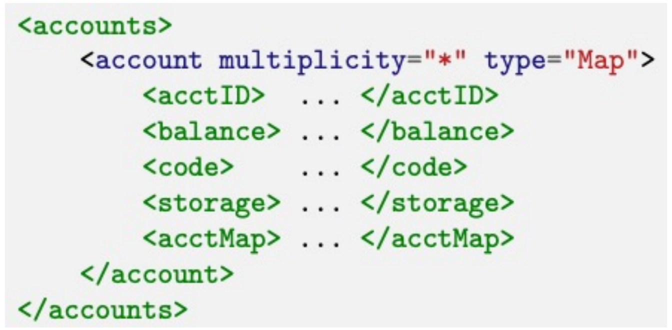
The state of EVM is broken into two components:

- Active Transaction (Smart Contract execution)



```
       <op> MSTORE 8 75                            </op>
       <id> ...                                    </id>
      <gas> 98422223                               </gas>
 <gasPrice> 1000000                                </gasPrice>
       <pc> 13                                     </pc>
  <program> ... 13 |-> MSTORE ...                  </program>
<wordStack> 7 : 9 : ...                            </wordStack>
 <localMem> ... 8 |-> _ ... 39 |-> _ ...           </localMem>
```

- Network as a whole (Account information)



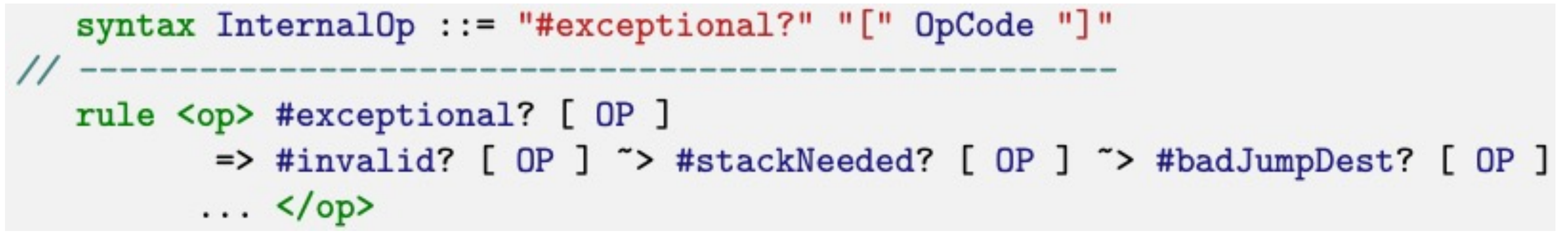
```
<accounts>
    <account multiplicity="*" type="Map">
        <acctID>  ... </acctID>
        <balance> ... </balance>
        <code>    ... </code>
        <storage> ... </storage>
        <acctMap> ... </acctMap>
    </account>
</accounts>
```

K Semantics of EVM

Exceptional States

Before attempting to execute an opcode KEVM check if one of several exceptional cases will happen:

- Is the next opcode one of the designated invalid or undefined opcodes?
- Will the stack over / under-flow upon executing the opcode?
- If the opcode is a JUMP* code, will it jump to a valid destination?

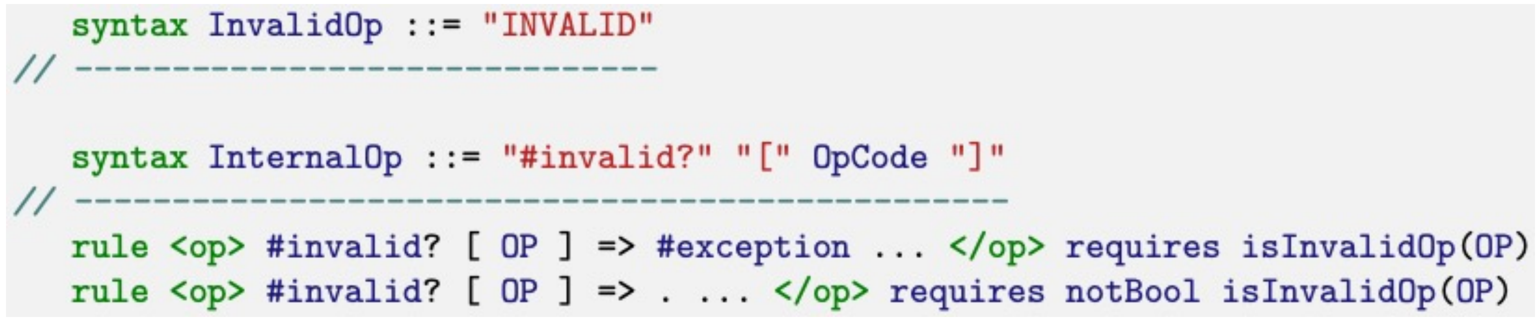


```
   syntax InternalOp ::= "#exceptional?" "[" OpCode "]"
// ----------------------------------------------------
   rule <op> #exceptional? [ OP ]
          => #invalid? [ OP ] ~> #stackNeeded? [ OP ] ~> #badJumpDest? [ OP ]
         ... </op>
```

K Semantics
of
EVM

Exceptional States

Example - #invalid? Definition

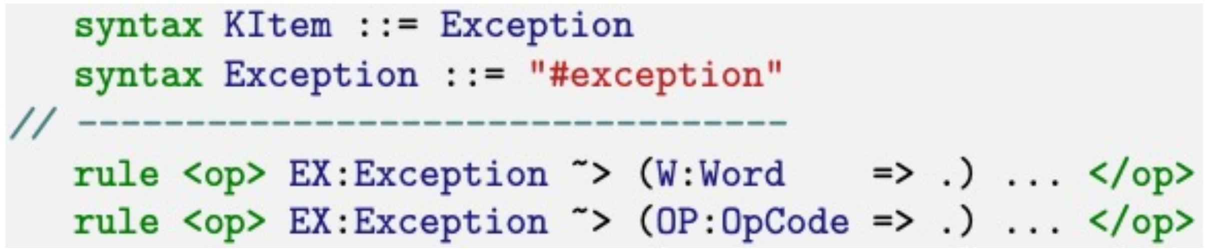


```
   syntax InvalidOp ::= "INVALID"
// ------------------------------

   syntax InternalOp ::= "#invalid?" "[" OpCode "]"
// ------------------------------------------------
   rule <op> #invalid? [ OP ] => #exception ... </op> requires isInvalidOp(OP)
   rule <op> #invalid? [ OP ] => . ... </op> requires notBool isInvalidOp(OP)
```

K Semantics of EVM

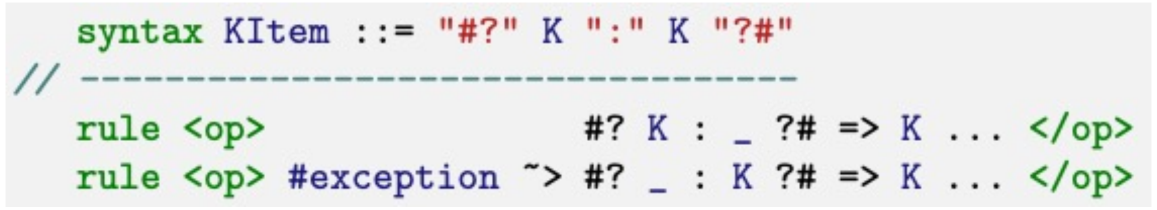
Control Flow

The exception consumes any ordinary Word or OpCode behind it on the cell, until it reaches a Kitem



```
   syntax KItem ::= Exception
   syntax Exception ::= "#exception"
// --------------------------------
   rule <op> EX:Exception ~> (W:Word    => .) ... </op>
   rule <op> EX:Exception ~> (OP:OpCode => .) ... </op>
```

By putting operators in sort KItem, we can use them to affect how an #exception is caught.



```
   syntax KItem ::= "#?" K ":" K "?#"
// ---------------------------------
   rule <op>                 #? K : _ ?# => K ... </op>
   rule <op> #exception ~> #? _ : K ?# => K ... </op>
```

#?_ : _?# - branching choice operator. Chose the first branch when no exception is thrown and the second when one is.

K Semantics
of
EVM

Reverting State

During execution, state may need to be partially or fully reverted when an exception is thrown.

KEVM supplies six operators to handle this:

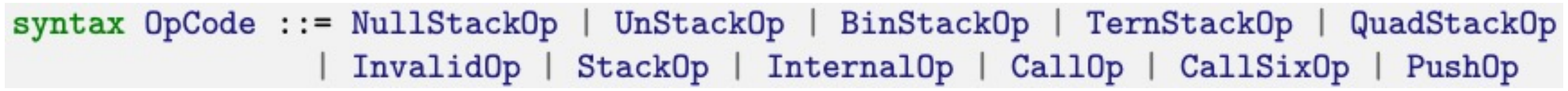
- #{push/pop/drop}CallStack
- #{push/pop/drop}WordStack

These operators can be used to save and restore copies of the execution and network state in the cells and , respectively.

K Semantics of EVM

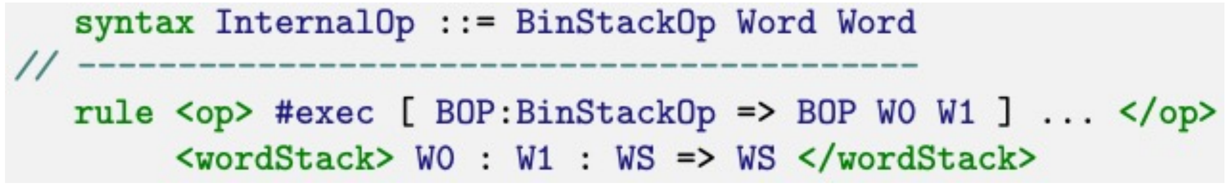
EVM OpCodes & Arguments Loading

EVM OpCodes
The opcodes in EVM are broken into several subsorts of OpCode when common behavior can be associated to that group of opcodes.



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

Arguments Loading
When an opcode is executed, the correct number of arguments are unpacked from the <wordStack>

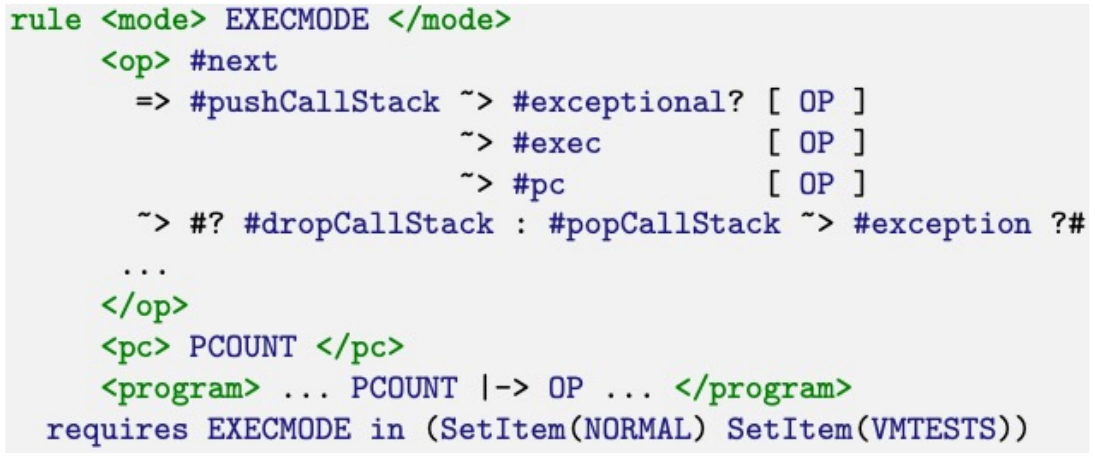


```
   syntax InternalOp ::= BinStackOp Word Word
// ------------------------------------------
   rule <op> #exec [ BOP:BinStackOp => BOP W0 W1 ] ... </op>
        <wordStack> W0 : W1 : WS => WS </wordStack>
```

K Semantics of EVM

Execution Cycle

1. Check if the opcode is exceptional given the current state (#exceptional?).
2. Perform the state update associated with the operator (#exec).
3. Increment the program counter to the correct value given the operator (#pc).



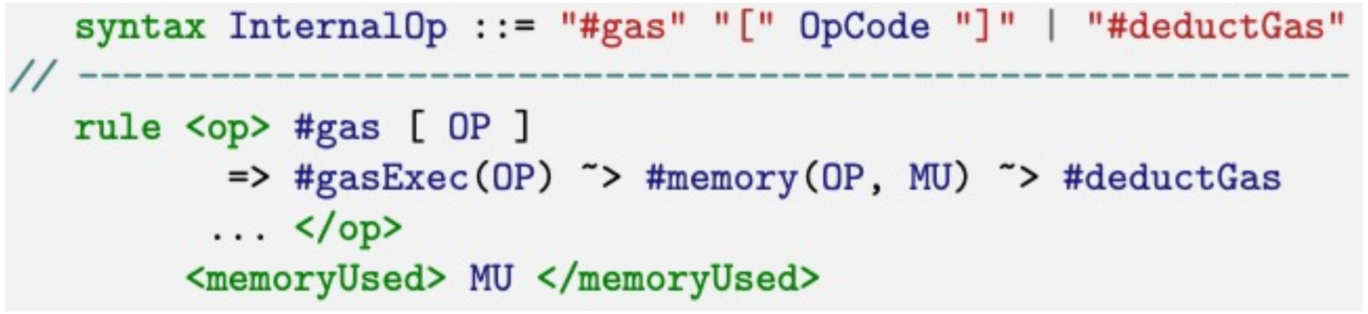
```
rule <mode> EXECMODE </mode>
     <op> #next
       => #pushCallStack ~> #exceptional? [ OP ]
                         ~> #exec         [ OP ]
                         ~> #pc           [ OP ]
       ~> #? #dropCallStack : #popCallStack ~> #exception ?#
      ...
     </op>
     <pc> PCOUNT </pc>
     <program> ... PCOUNT |-> OP ... </program>
  requires EXECMODE in (SetItem(NORMAL) SetItem(VMTESTS))
```

K Semantics of EVM

Gas Semantics

The #exec operator above is turned into two consecutive state update:

1. First, the #gas update is applied
2. Then updates to execution state are applied



```
   syntax InternalOp ::= "#gas" "[" OpCode "]" | "#deductGas"
// -----------------------------------------------------------
   rule <op> #gas [ OP ]
          => #gasExec(OP) ~> #memory(OP, MU) ~> #deductGas
         ... </op>
        <memoryUsed> MU </memoryUsed>
```

- Several fee schedules are provided.
- KEVM provide additional helper functions for more complicated gas calculation.

K Semantics
of
EVM

OpCode Semantics

- Operator #exec places the opcode directly at the front of the cell with its arguments from the loaded
- the semantics of ADD:

```
rule <op> ADD W0 W1 => W0 +Word W1 ~> #push ... </op>
```

  - _+Word_ is a word operation supplied in the semantics which lifts the native K +Int operation and ensures the correct modulus-256 semantics are implemented for EVM.
  - Reminder - behind the addition, there’s a helper #push which places the preceding word on the front of the <wordStack>

Semantic Evaluation

Semantic Evaluation

KEVM evaluation focused on measuring:

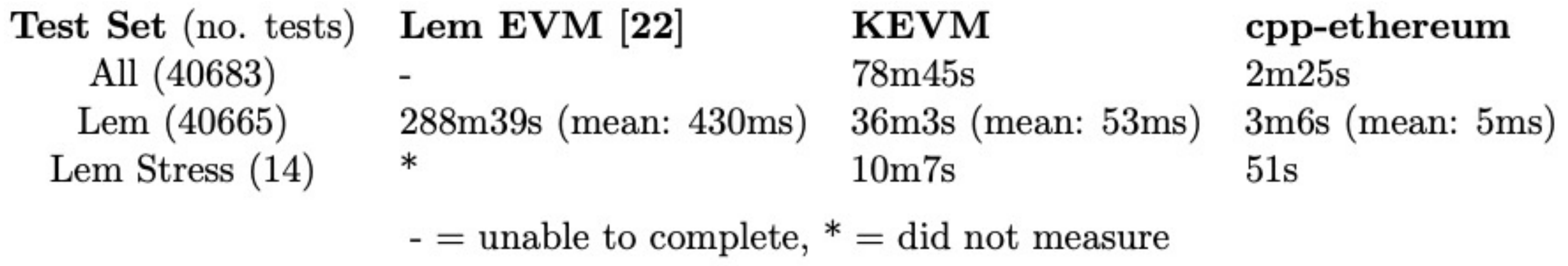
1. **Correctness** – As Consensus-critical software, implementations of the EVM are held to a high standard

2. **Performance** – important for real-world application and usability

3. **Extensibility** – KEVM was built with extensibility in mind

Semantic Evaluation

Correctness & Performance



Test Set (no. tests)	**Lem EVM [22]**	**KEVM**	**cpp-ethereum**
All (40683)	-	78m45s	2m25s
Lem (40665)	288m39s (mean: 430ms)	36m3s (mean: 53ms)	3m6s (mean: 5ms)
Lem Stress (14)	*	10m7s	51s

- = unable to complete, * = did not measure

Table 2: Completeness and total execution time of existing executable semantics

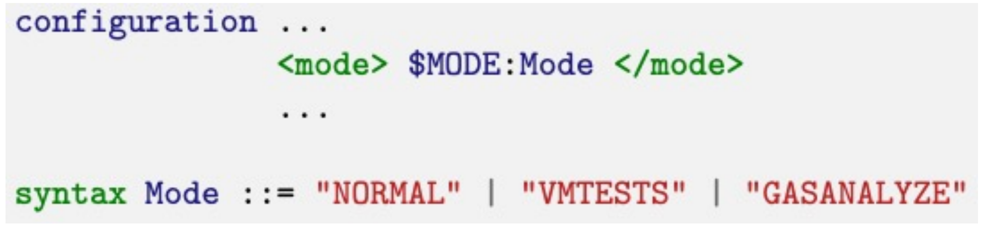
- As shown in the comparison, the automatically extracted interpreter for KEVM outperforms the currently available formal executable EVM semantics
- In addition to handling the entirety of the test-suite, the KEVM semantics outperforms the Lem semantics by more than eight times
- Compared to the C++ implementation KEVM performs favorably, coming in roughly 30 times slower than the reference implementation on all 40,683 tests

These numbers are very promising for an early version of an automatically generated, formally derived interpreter, and substantiates the practicality of KEVM approach.

Semantic Evaluation

Extensibility

- The simple imperative language with control-flow supplied via exceptions and conditional branching allows to add more primitives for extending KEVM:
  - For example, KEVM define another primitive #execTo which takes a set of opcodes and calls #next until we reach one of the opcodes in the set.
- Execution is parametric over an extra <mode> cell
  - Currently, three modes are supported: NORMAL execution, VMTESTS execution, and GASANALYZE mode.



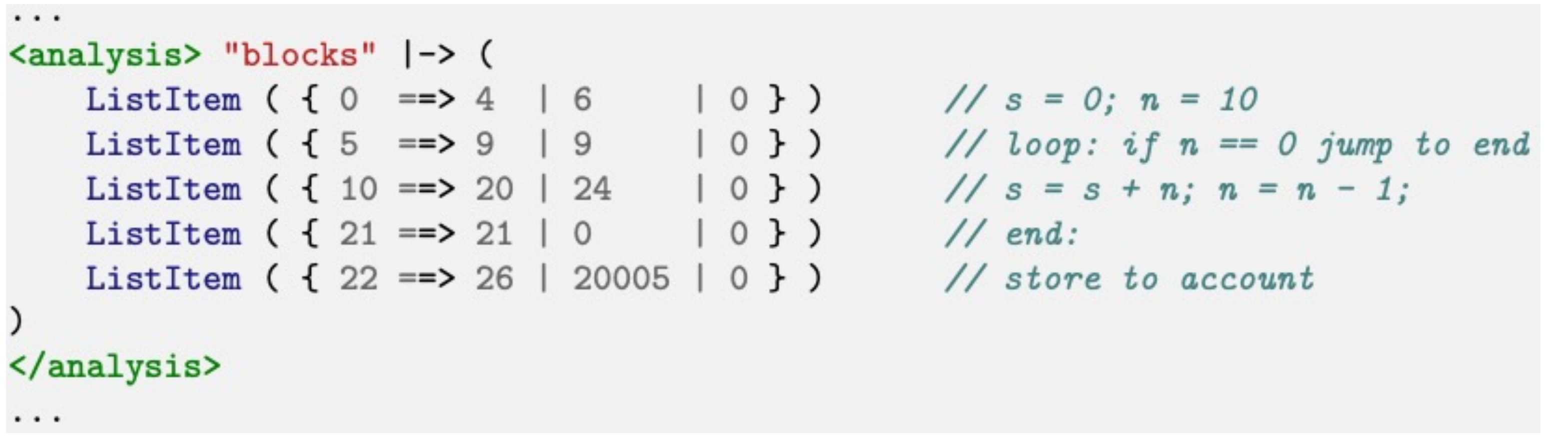
```
configuration ...
              <mode> $MODE:Mode </mode>
              ...

syntax Mode ::= "NORMAL" | "VMTESTS" | "GASANALYZE"
```

Semantic Evaluation

Extensibility – Gas Analysis

For many contracts, functional correctness is dependent upon enough gas being supplied at the beginning of execution.

By calling krun with the option -cMODE=GASANALYSE, KEVM produce the following output (on program that sum the numbers from 1 to 10):



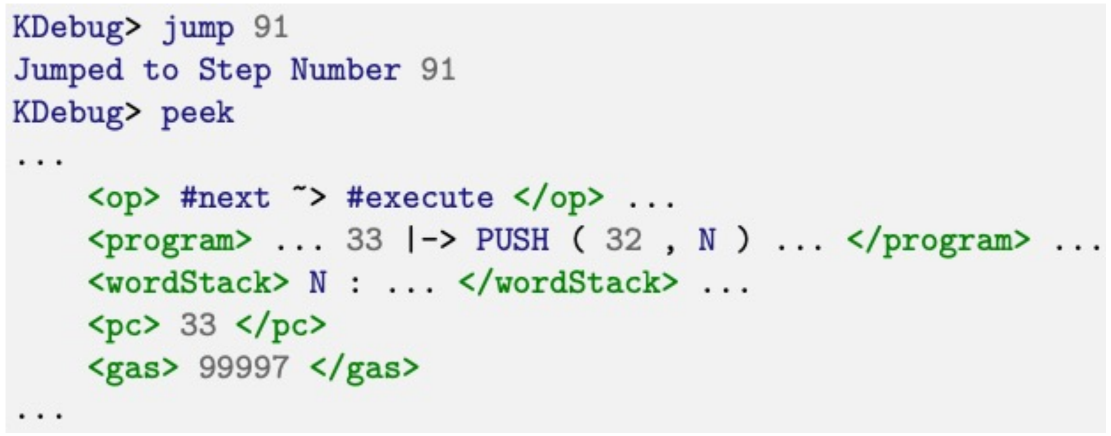
```
...
<analysis> "blocks" |-> (
    ListItem ( { 0  ==> 4  | 6     | 0 } )      // s = 0; n = 10
    ListItem ( { 5  ==> 9  | 9     | 0 } )      // loop: if n == 0 jump to end
    ListItem ( { 10 ==> 20 | 24    | 0 } )      // s = s + n; n = n - 1;
    ListItem ( { 21 ==> 21 | 0     | 0 } )      // end:
    ListItem ( { 22 ==> 26 | 20005 | 0 } )      // store to account
)
</analysis>
...
```

This GASANALYZE mode only added 52 lines to the semantics.

Derived Analysis Tools

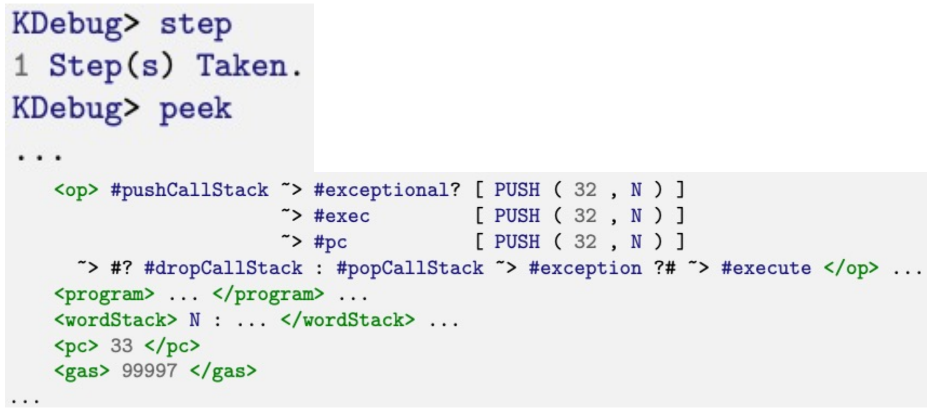
Semantic Debugger

Adding the --debugger option to the command krun drops the user into the KDebug shell.

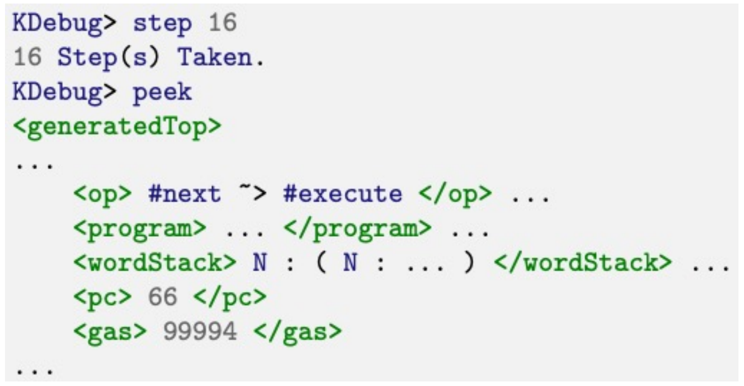


```
KDebug> jump 91
Jumped to Step Number 91
KDebug> peek
...
    <op> #next ~> #execute </op> ...
    <program> ... 33 |-> PUSH ( 32 , N ) ... </program> ...
    <wordStack> N : ... </wordStack> ...
    <pc> 33 </pc>
    <gas> 99997 </gas>
...
```

Taking one step, we see that a full execution cycle for the next opcode has been loaded & Taking another 16 steps, the opcode is fully executed:



```
KDebug> step
1 Step(s) Taken.
KDebug> peek
...
   <op> #pushCallStack ~> #exceptional? [ PUSH ( 32 , N ) ]
                       ~> #exec         [ PUSH ( 32 , N ) ]
                       ~> #pc           [ PUSH ( 32 , N ) ]
     ~> #? #dropCallStack : #popCallStack ~> #exception ?# ~> #execute </op> ...
   <program> ... </program> ...
   <wordStack> N : ... </wordStack> ...
   <pc> 33 </pc>
   <gas> 99997 </gas>
...
```



```
KDebug> step 16
16 Step(s) Taken.
KDebug> peek
<generatedTop>
...
    <op> #next ~> #execute </op> ...
    <program> ... </program> ...
    <wordStack> N : ( N : ... ) </wordStack> ...
    <pc> 66 </pc>
    <gas> 99994 </gas>
...
```

Derived Analysis Tools

Program Verifier

One particularly useful formal analysis tool developed for K is the Reachability Logic prover.

This prover accepts as input a K definition and a set of logical reachability claims to prove.

The prover then attempts to automatically prove the reachability theorems over the language's execution space, assuming the semantics.

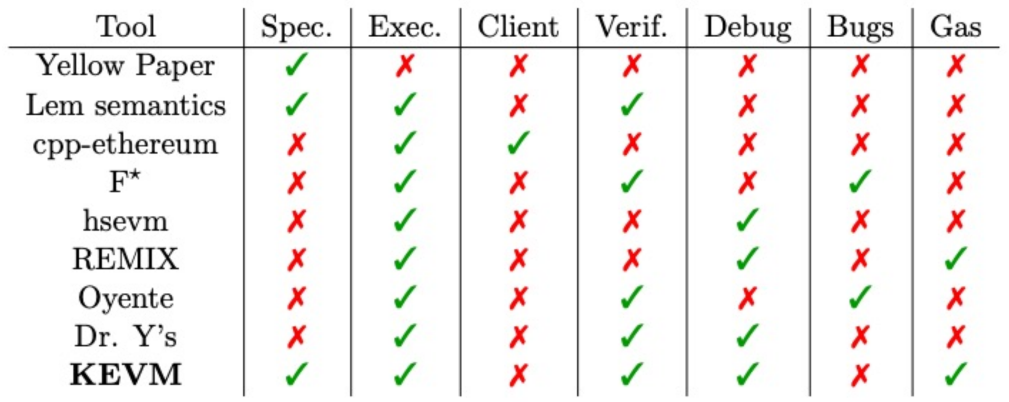
Summary

Feature Comparison Overview

The tools produced in the Ethereum community are meant to fill a variety of purposes, many of which are also able to be accomplished directly from our executable semantics.

For the implementations we compare, we ask the following questions about the tools provided:

- Spec.: Suitable as a formal specification of the EVM language?
- Exec.: Executable on concrete tests?
- Client: Implements a full Ethereum client?
- Verif.: Used to verify properties about EVM programs?
- Debug: Supplies interactive EVM debugger?
- Bugs: Heuristic-based tools for finding common issues in smart contracts?
- Gas: Tools for analyzing gas complexity of an EVM program?



Tool	Spec.	Exec.	Client	Verif.	Debug	Bugs	Gas
Yellow Paper	✓	✗	✗	✗	✗	✗	✗
Lem semantics	✓	✓	✗	✓	✗	✗	✗
cpp-ethereum	✗	✓	✓	✗	✗	✗	✗
F*	✗	✓	✗	✓	✗	✓	✗
hsevm	✗	✓	✗	✗	✓	✗	✗
REMIX	✗	✓	✗	✗	✓	✗	✓
Oyente	✗	✓	✗	✓	✗	✓	✗
Dr. Y's	✗	✓	✗	✓	✓	✗	✗
KEVM	✓	✓	✗	✓	✓	✗	✓

Questions?

Thank you for listening!