



Formal Design, Implementation and Verification of Blockchain Languages

Grigore Rosu

University of Illinois at Urbana-Champaign

President & CEO, Runtime Verification Inc.

ICBC'19, May 17, 2019

Think “execute some code given, publicly visible code, with shared state”

Transaction is broadcast, then “validated” by re-executing it on many “nodes” in many languages (e.g., C++, Java, JavaScript, Python, etc.)

Validated transactions are then deployed by all nodes locally...

Some transactions add new code to the blockchain, called “smart contracts”, which can be executed by other transactions.

Transactions are grouped into blocks, appending to the previous block, irreversibly, to form a public “ledger” or “history” or “blockchain”.



On the network, transactions are combined with other transactions to create a new block of data for the next block.



Has no physical form and exists only in the network.

Hackers have huge incentives to exploit any bugs in smart contracts or underlying infrastructure. In the end, all code is public, can be invoked by anybody, and can irreversibly change the history (e.g., steal your money).

Smart Contract Snippet (ERC20)

(one of the ~40,000 Ethereum ERC20s)

Written in Solidity:

```
function transfer(address _to, uint256 _value) returns (bool success) {  
  
    ...  
  
    if (_value == 0) { return false; }  
  
    uint256 fromBalance = balances[msg.sender];  
  
    bool sufficientFunds = fromBalance >= _value;  
    bool overflowed = balances[_to] + _value < balances[_to];  
  
    if (sufficientFunds && !overflowed) {  
        balances[msg.sender] -= _value;  
        balances[_to] += _value;  
  
        Transfer(msg.sender, _to, _value);  
        return true;  
    } else { return false; }  
}
```

ERC20 does not
state that...

There should be
no overflow when
self-transfer...

Attack 11

Parity Multisig Hacked. Again

Yesterday, Parity Multisig Wallet was hacked again:

<https://paritytech.io/blog/security-alert.html>

“This means that currently no funds can be moved out of the [ANY Parity] multisig wallets”

A lot of people/companies/ICOs are using Parity-generated multisig wallets.

About **\$300M** is frozen and (probably) lost forever.

Disclaimer: I lost little money (about \$1000) but my friends lost about \$300K.

Who hacked it?

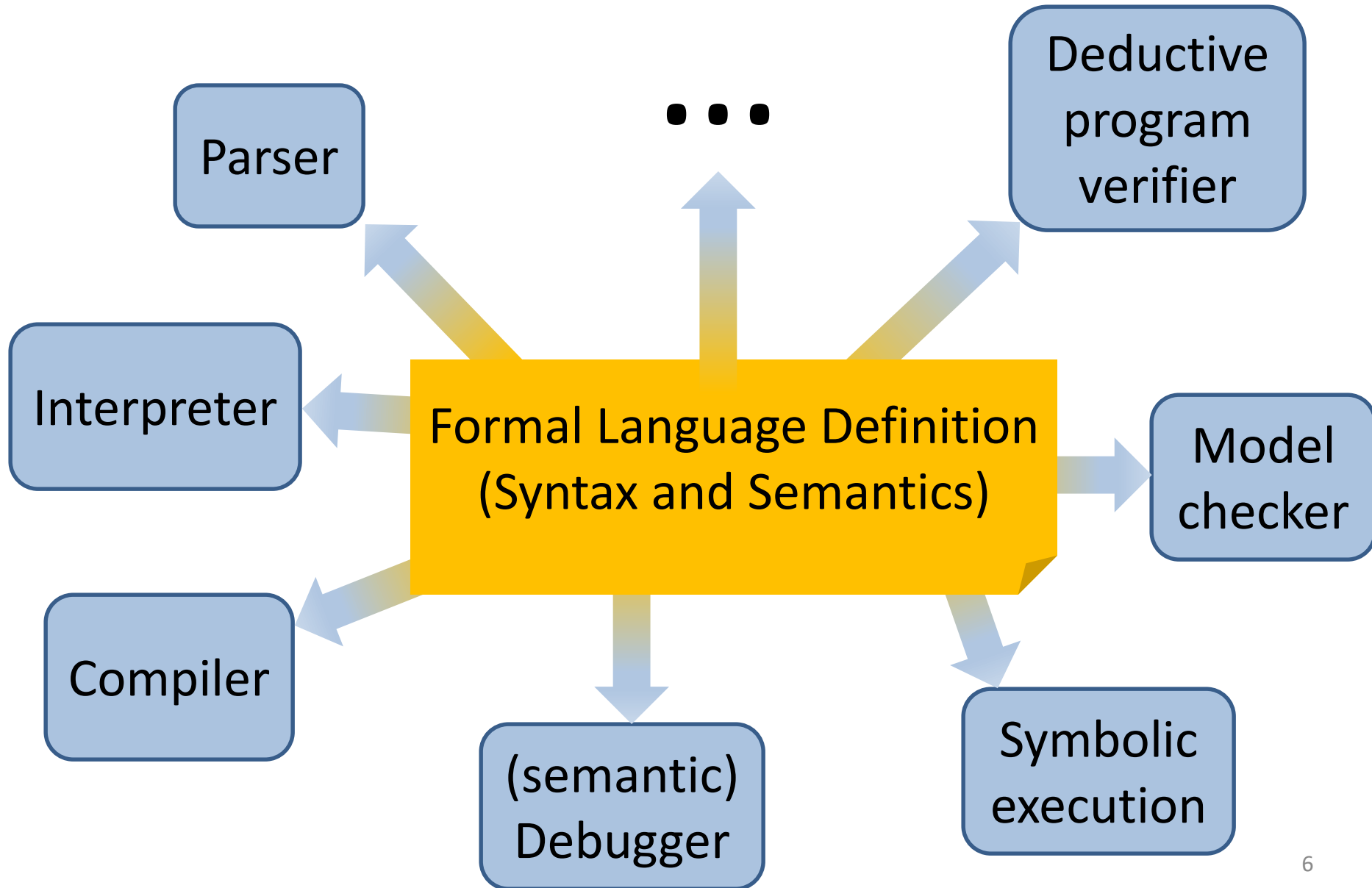
Some guy with a nickname @devops199 (not a member of the Parity team) and an “empty” github account. His Ethereum address is 0xae7168Deb525862f4FEe37d987A971b385b96952 and he has successfully verified it.

Blockchain. This first post will focus on exactly the attacker stole all the money in the DAO.

What Can We Do About This?

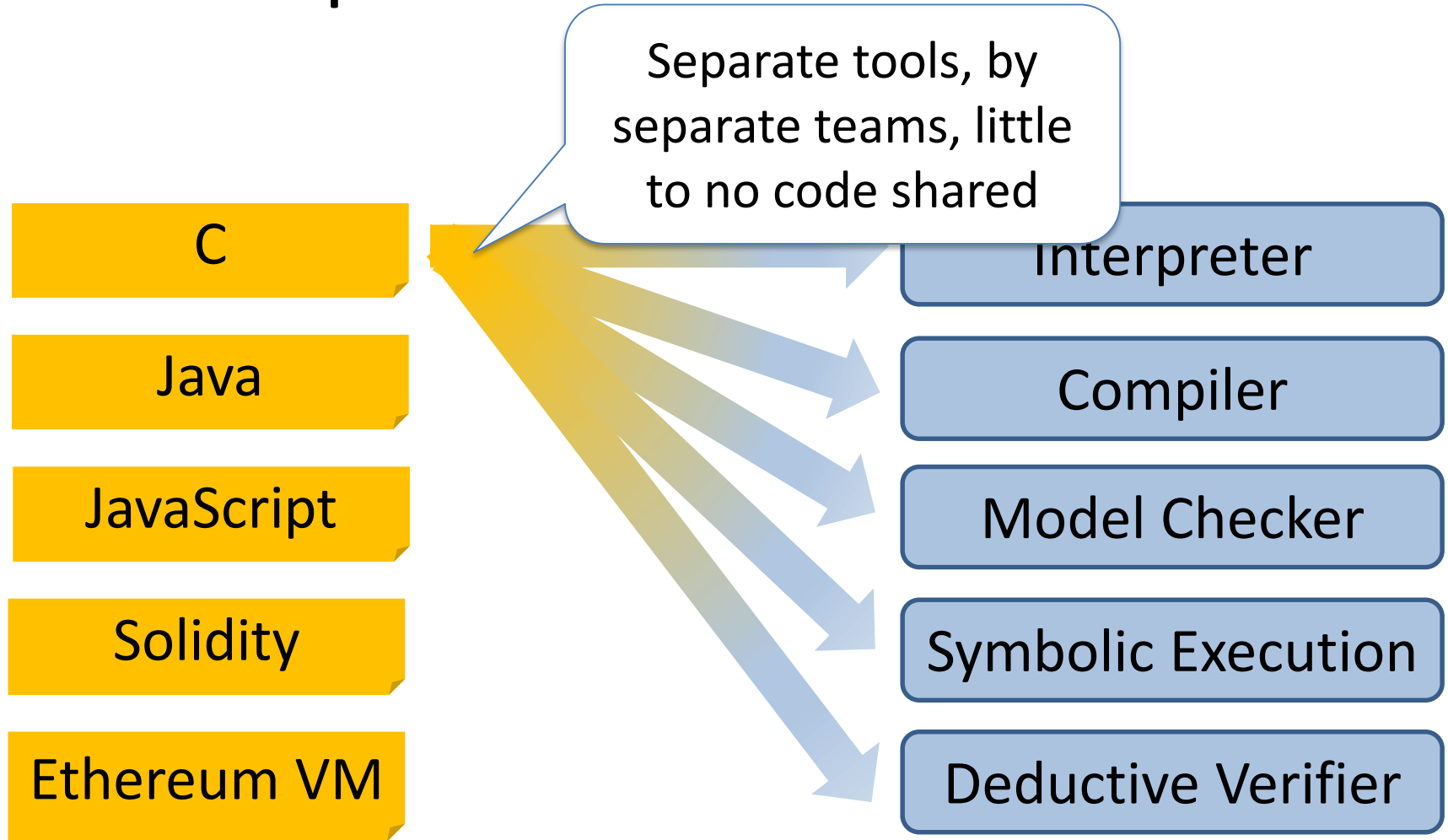
- More specifically, what can we do about the **execution environment**, to increase security?
 - Unacceptable to build this complex and disruptive technology with poorly designed VMs and languages!
- **Ideal scenario feasible, stop compromising!**
 - **Everything must be rigorously designed**, using formal methods. Implementations must be **provably correct**!
 - Nodes: **provably correct** VMs or interpreters
 - Smart contracts: use **well-designed programming languages**, with **provably correct** compilers or interpreters
 - Verification: Smart contracts **provably correct** wrt their specs

Ideal Language Framework Vision



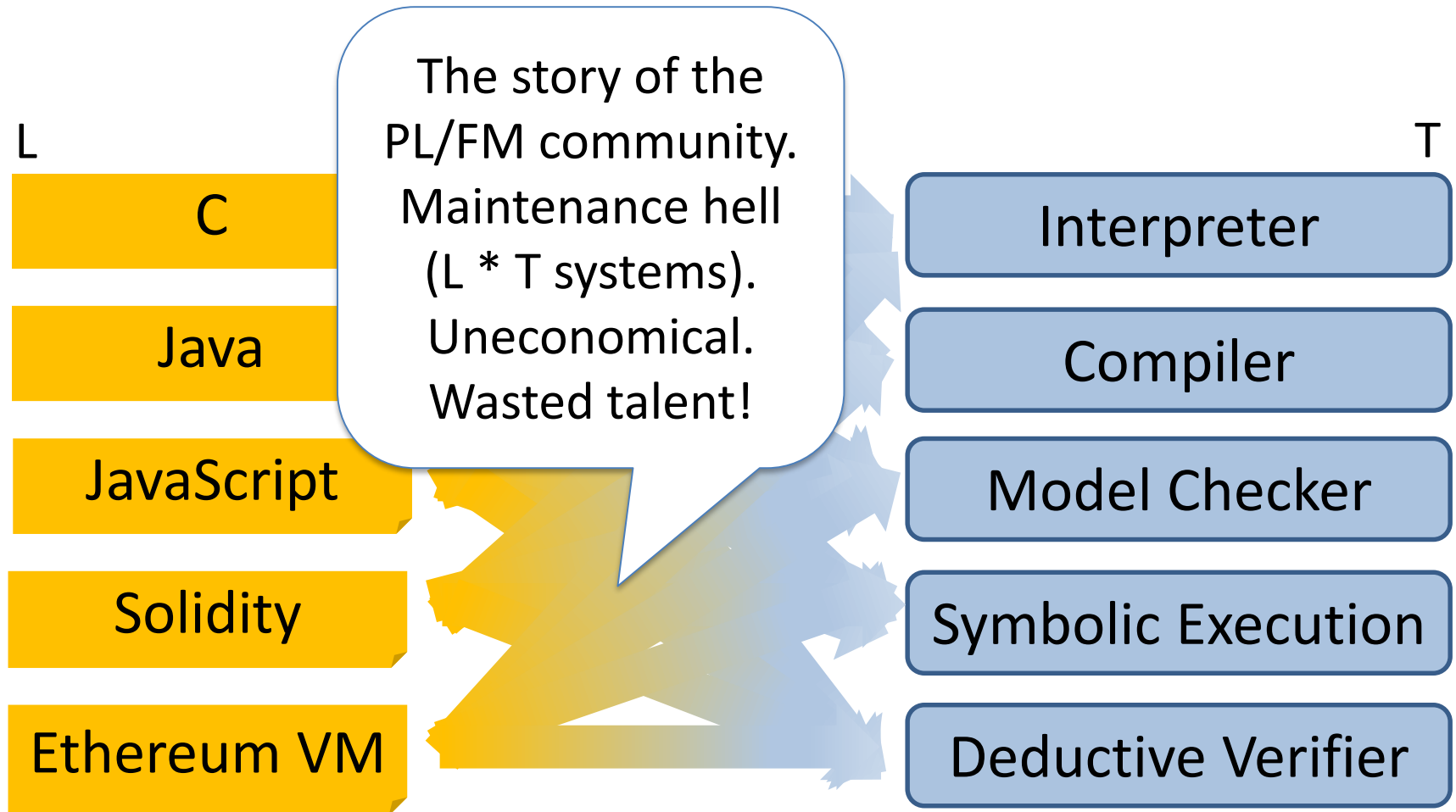
Current State-of-the-Art

- Sharp Contrast to Ideal Vision -

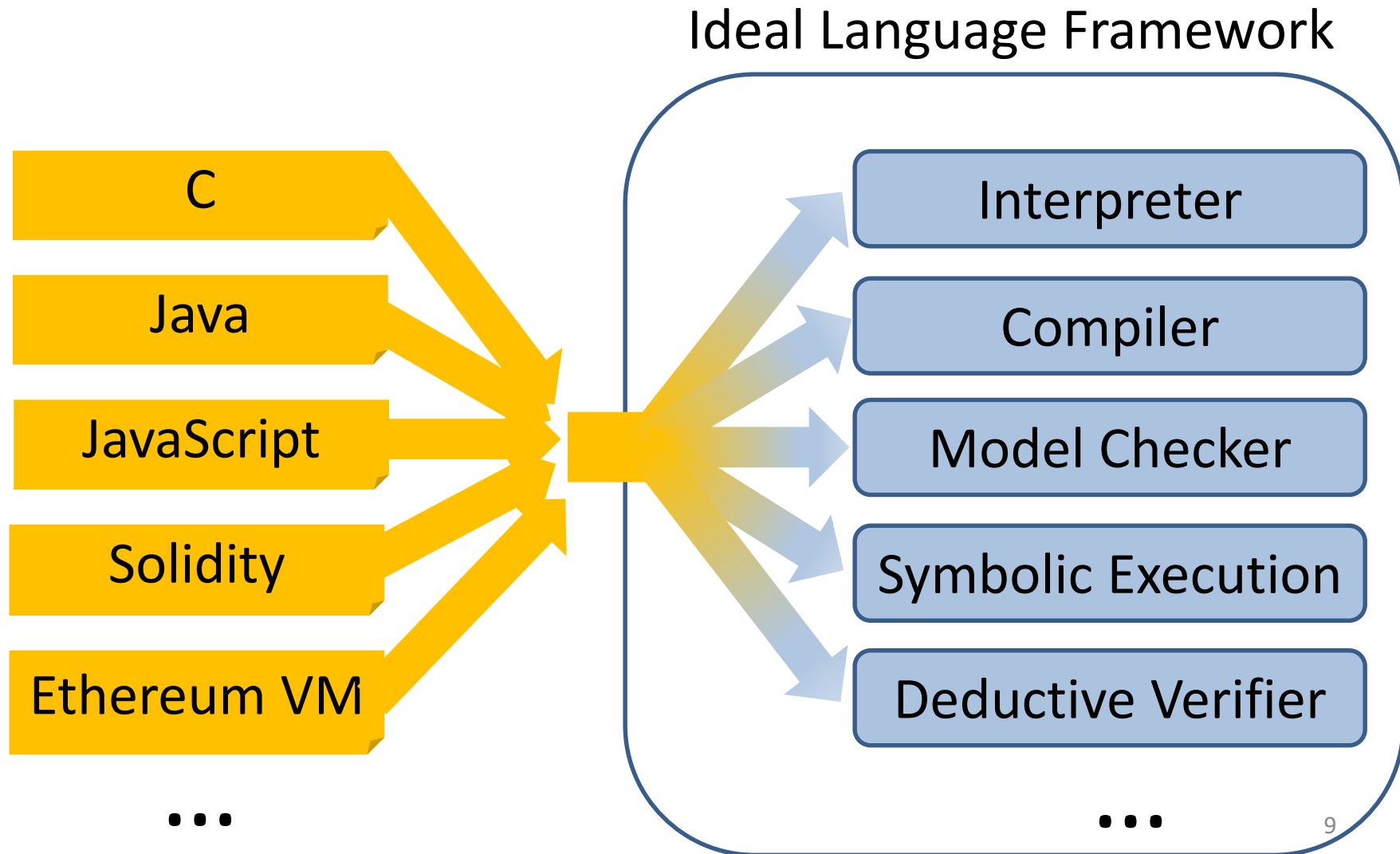


Current State-of-the-Art

- Sharp Contrast to Ideal Vision -



How It Should Be



Our Attempt: the K Framework

<http://kframework.org>

- We tried various semantic styles, for >15y and >100 top-tier conference and journal papers:
 - Small-/big-step SOS; Evaluation contexts; Abstract machines (CC, CK, CEK, SECD, ...); Chemical abstract machine; Axiomatic; Continuations; Denotational;...
- But each of the above had limitations
 - Especially related to modularity, notation, verification
- K framework initially *engineered*: keep advantages and avoid limitations of various semantic styles
 - Then theory came

Complete K Definition of KernelC

```

MODULE KERNELC-SYNTAX
IMPORTS K-LATEX+PL-ID+PL-INT
SYNTAX Exp ::= Exp * Exp [strict]
          DeclId
          Id
          Int
          Exp * Exp [strict]
          Exp ++
          Exp == Exp [strict]
          Exp != Exp [strict]
          Exp < Exp [strict]
          Exp % Exp [strict]
          Exp < Exp [strict]
          ! Exp
          Exp && Exp
          Exp ? Exp : Exp
          Exp || Exp
          printf("%d", Exp) [strict]
          scanf("%d", & Exp) [strict]
          scanf("%d", Exp) [strict]
          NULL
          PointerId
          (int*)malloc( Exp * sizeof(int)) [strict]
          free( Exp ) [strict]
          * Exp [strict]
          Exp [ Exp ]
          Exp = Exp [strict2]
          Id ( List(Exp) ) [strict2]
          Id ( )
          random()
          srand( Exp ) [strict]
SYNTAX Stmt ::= Exp ; [strict]
          { }
          { StmtList }
          if( Exp ) Stmt
          if( Exp ) Stmt else Stmt [strict1]
          while( Exp ) Stmt
          return Exp ; [strict]
          DeclId List(DeclId) { StmtList }
          #include< StmtList >
SYNTAX StmtList ::= StmtList StmtList
          Stmt
SYNTAX Pgm ::= StmtList
SYNTAX Id ::= main
SYNTAX PointerId ::= * PointerId [ditto]
          | Id
SYNTAX DeclId ::= int Exp
          | void PointerId
SYNTAX StmtList ::= stdio.h
          | stdlib.h
SYNTAX List(Bottom) ::= List(Bottom) , List(Bottom) [assoc hybrid id: ( ) strict]
          | ( )
          | Bottom
SYNTAX List(PointerId) ::= List(PointerId) , List(PointerId) [ditto]
          | List(Bottom)
          | PointerId
SYNTAX List(DeclId) ::= List(DeclId) , List(DeclId) [ditto]
          | DeclId
SYNTAX List(Exp) ::= List(Exp) , List(Exp) [ditto]
          | Exp
          | List(DeclId)
          | List(PointerId)
END MODULE

```

```

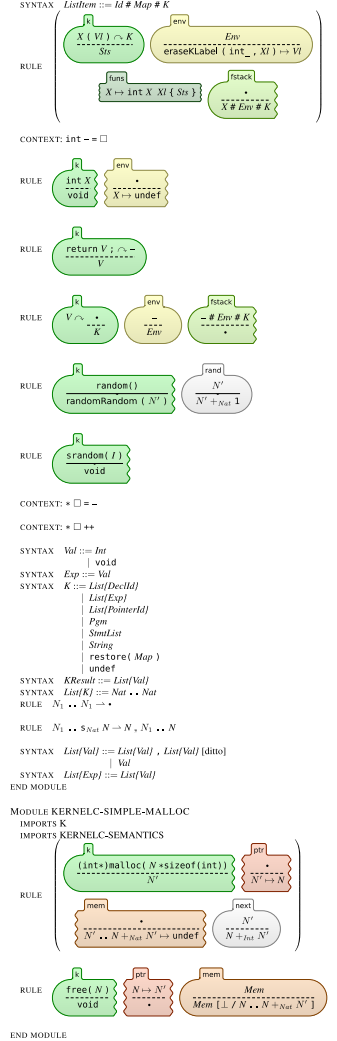
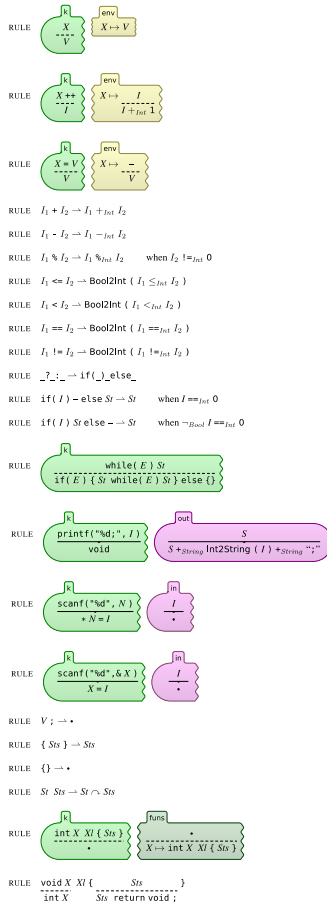
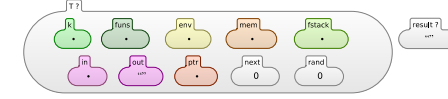
MODULE KERNELC-DESUGARED-SYNTAX
IMPORTS K-LATEX
IMPORTS KERNELC-SYNTAX
MACRO ! E = E ? 0 : 1
MACRO E1 && E2 = E1 ? E2 : 0
MACRO E1 || E2 = E1 ? 1 : E2
MACRO if( E ) St = if( E ) St else { }
MACRO NULL = 0
MACRO I ( ) = I ( ( ) )
MACRO int * PointerId = int PointerId
MACRO #include< Stmts > = Stmts
MACRO E1 [ E2 ] = * E1 + E2
MACRO scanf("%d", & * E) = scanf("%d", E)
MACRO int * PointerId = E = int PointerId = E
MACRO int X = E ; = int X ; X = E ;
MACRO stdio.h = { }
MACRO stdlib.h = { }
END MODULE

```

```

MODULE KERNELC-SEMANTICS
IMPORTS K-SHARED
IMPORTS K-KERNELC-DESUGARED-SYNTAX+PL-CONVERSION+PL-RANDOM
CONFIGURATION:

```



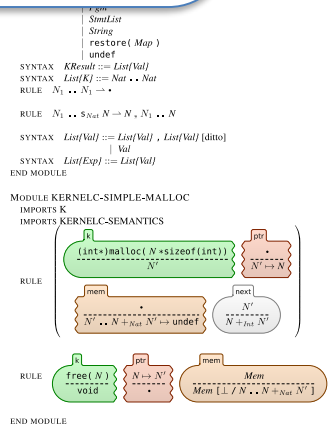
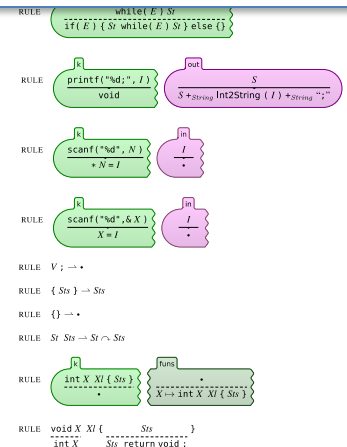
```

MODULE KERNEL-SYNTAX
IMPORTS K-LATEX+PL-ID+PL-INT
SYNTAX  Exp ::= Exp + Exp [strict]
        | DeclId
        | Id
        | Int
        | Exp * Exp [strict]
        | Exp ++
        | Exp == Exp [strict]
        | Exp != Exp [strict]
        | Exp <= Exp [strict]
        | Exp < Exp [strict]
        | Exp > Exp [strict]
        | Exp >= Exp [strict]
        | Exp && Exp
        | Exp ? Exp : Exp
        | Exp [! Exp]
        | printf("wd", , Exp) [strict]
        | scanf("wd", &Exp)
        | scanf("wd", , Exp) [strict]
        | NULL
        | pointerf
          (int* malloc(Exp + sizeof(int)) [strict]
           | Tree(Exp) [strict]
           | *Exp [strict]
           | Exp [! Exp])
        | Exp = Exp [strict?2]
        | Id (List(Exp) ) [strict?2]
        | Id ( )
        | random()
        | srandom(Exp) [strict]
SYNTAX  Smt ::= Exp [strict]
        | { }
        | { SmtList }
        | if (Exp) Smt
        | if (Exp) Smt else Smt [strict?1]
        | while (Exp) Smt
        | return Exp ; [strict]
        | DeclId List(DeclId) ( SmtList )
        | #include: SmtList
SYNTAX  SmtList ::= SmtList SmtList
        | Smt
SYNTAX  Pgm ::= SmtList
SYNTAX  Id ::= main
SYNTAX  PointerId ::= * PointerId [ditto]
        | Id
SYNTAX  DeclId ::= int Exp
        | void PointerId
SYNTAX  SmtList ::= stdio.h
        | stdlib.h
SYNTAX  List(Bottom) ::= List(Bottom) , List(Bottom) [assoc hybrid id: ( ) strict]
        | ( )
        | Bottom
SYNTAX  List(PointerId) ::= List(PointerId) , List(PointerId) [ditto]
        | PointerId
SYNTAX  List(DeclId) ::= List(DeclId) , List(DeclId) [ditto]
        | DeclId
SYNTAX  List(Exp) ::= List(Exp) , List(Exp) [ditto]
        | Exp
        | List(DeclId)
        | List(PointerId)
END MODULE

MODULE KERNEL-DESUGARED-SYNTAX
IMPORTS K-LATEX
IMPORTS KERNEL-SYNTAX
MACRO  ! E = E ? 0 : 1
MACRO  E1 && E2 = E1 ? E2 : 0
MACRO  E1 || E2 = E1 ? 1 : E2
MACRO  if ( E ) S = if ( E ) S else { }
MACRO  NULL = 0
MACRO  I ( ) = I ( ) ( )
MACRO  int * PointerId = int PointerId
MACRO  #include: Smts > = Smts
MACRO  E1 [ E2 ] = * E1 + E2
MACRO  scanf("wd", &* E) = scanf("wd", , E)
MACRO  int * PointerId = E = int PointerId = E
MACRO  int X = E ; = int X ; X = E ;
MACRO  stdio.h = { }
MACRO  stdlib.h = { }

```

$$\text{SYNTAX} \quad \textit{Exp} ::= \dots$$

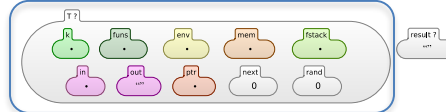
$$\quad \quad \quad | \textit{Exp} = \textit{Exp} \text{ [strict(2)]}$$


```

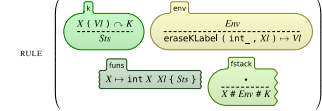
MODULE KERNEL-SYNTAX
IMPORTS K+LATEX+PL-ID+PL-INT
SYNTAX Exp ::= Exp + Exp [strict]
          | Decid
          | Id
          | Int
          | Exp - Exp [strict]
          | Exp **
          | Exp == Exp [strict]
          | Exp = Exp [strict]
          | Exp <= Exp [strict]
          | Exp < Exp [strict]
          | Exp % Exp [strict]
          | ! Exp
          | Exp && Exp
          | Exp ? Exp : Exp
          | Exp || Exp
          | print("sd", Exp) [strict]
          | scanf("sd", Exp) [strict]
          | scanf("sd", Exp) [strict]
          | NULL
          | PointerId
          | (lvs...)
          | If

```

```
MODULE KERNELC-SEMANTICS
IMPORTS K-SHARED
IMPORTS K+KERNELC-DESUGARED-SYNTAX+PL-CONVERSION+PL-RANDOM
CONFIGURATION:
```



SYNTAX $ListItem ::= Id \# Map \# K$



CONTEXT: `int - = □`

RULE

k	env
$\frac{\text{int } X}{\text{void}}$	$\frac{\cdot}{X \mapsto \text{undef}}$

RULE return $V: \ominus -$

SYNTAX

SYNTAX

SYNTAX

SYNTAX

SYNTAX

OXLEY ET AL.

SYNTAX

END MODU

MODULE K

MACRO

MACRO

STACRO

MACRO

MACRO

MACRO

MACRO

MACRO

MACRO

MACRO

END MODU

RULE $\frac{\text{void } X \text{ } XI \{ \quad \quad \quad Sfs \quad \quad \quad }{int \text{ } X \quad \quad \quad Sfs \text{ return void ;}}$

END MODULE

Complete K Definition of KernelC

```

MODULE KERNELC-SYNTAX
IMPORTS K-LATEX+PL-ID+PL-INT
SYNTAX Exp ::= Exp * Exp [strict]
          DeclId
          Id
          Int
          Exp * Exp [strict]
          Exp ++
          Exp == Exp [strict]
          Exp != Exp [strict]
          Exp < Exp [strict]
          Exp % Exp [strict]
          ! Exp
          Exp && Exp
          Exp ? Exp : Exp
          Exp || Exp
          printf("%d", Exp) [strict]
          scanf("%d", & Exp) [strict]
          scanf("%d", Exp) [strict]
          NULL
          PointerId
          (int*)malloc( Exp * sizeof(int)) [strict]
          free( Exp ) [strict]
          * Exp [strict]
          Exp [ Exp ]
          Exp = Exp [strict2]
          Id ( List(Exp) ) [strict2]
          Id ( )
          random()
          srand( Exp ) [strict]
SYNTAX Stmt ::= Exp ; [strict]
          { }
          { StmtList }
          if( Exp ) Stmt
          if( Exp ) Stmt else Stmt [strict1]
          while( Exp ) Stmt
          return Exp ; [strict]
          DeclId List(DeclId) { StmtList }
          #include< StmtList >
SYNTAX StmtList ::= StmtList StmtList
          Stmt
SYNTAX Pgm ::= StmtList
SYNTAX Id ::= main
SYNTAX PointerId ::= * PointerId [dito]
          Id
SYNTAX DeclId ::= int Exp
          void PointerId
          StmtList ::= stdio.h
          stdlib.h
SYNTAX List(Bottom) ::= List(Bottom) , List(Bottom) [assoc hybrid id: ( ) strict]
          ( )
          Bottom
SYNTAX List(PointerId) ::= List(PointerId) , List(PointerId) [dito]
          List(Bottom)
          PointerId
SYNTAX List(DeclId) ::= List(DeclId) , List(DeclId) [dito]
          DeclId
          List(Bottom)
SYNTAX List(Exp) ::= List(Exp) , List(Exp) [dito]
          Exp
          List(DeclId)
          List(PointerId)
END MODULE

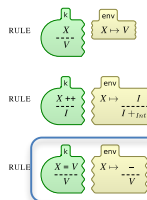
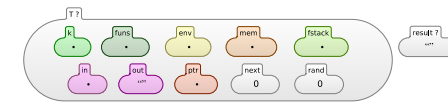
```

```

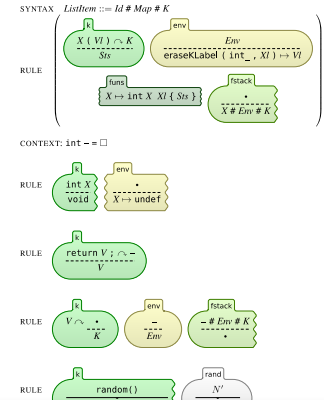
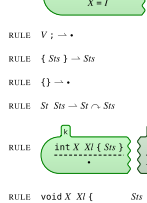
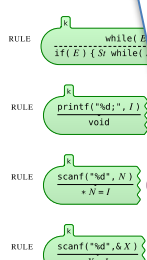
MODULE KERNELC-DESUGARED-SYNTAX
IMPORTS K-LATEX
IMPORTS KERNELC-SYNTAX
MACRO ! E = E ? 0 : 1
MACRO E1 && E2 = E1 ? E2 : 0
MACRO E1 || E2 = E1 ? 1 : E2
MACRO if( E ) St = if( E ) St else { }
MACRO NULL = 0
MACRO I ( ) = I ( ( ) )
MACRO int * PointerId = int PointerId
MACRO #include< Stmts > = Stmts
MACRO E1 [ E2 ] = * E1 + E2
MACRO scanf("%d", & * E) = scanf("%d", E)
MACRO int * PointerId = E = int PointerId = E
MACRO int X = E ; = int X ; X = E ;
MACRO stdio.h = { }
MACRO stdlib.h = { }
END MODULE

```

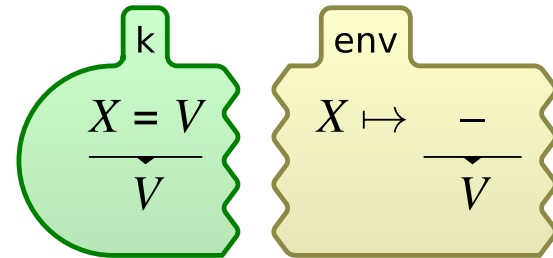
MODULE KERNELC-SEMANTICS
IMPORTS K-SHARED
IMPORTS K-KERNELC-DESUGARED-SYNTAX+PL-CONVERSION+PL-RANDOM
CONFIGURATION:



$I_1 + I_2 \rightarrow I_1 +_{int} I_2$
 $I_1 \cdot I_2 \rightarrow I_1 \cdot I_2$
 $I_1 \% I_2 \rightarrow I_1 \% I_2$ when $I_2 \neq 0$
 $I_1 \leq I_2 \rightarrow \text{Bool}$ ($I_1 \leq_{int} I_2$)
 $I_1 < I_2 \rightarrow \text{Bool2}$ ($I_1 <_{int} I_2$)
 $I_1 = I_2 \rightarrow \text{Bool2}$
 $I_1 ! = I_2 \rightarrow \text{Bool2}$
 $?_1 ; \dots \rightarrow \text{if}(?) \text{ else } \dots$
 $\text{if}(I) - \text{else } St \rightarrow St$
 $\text{if}(I) - \text{else } St \rightarrow St$



Semantic rules given contextually



rule

$\langle k \rangle X = V \Rightarrow V \dots \langle /k \rangle$

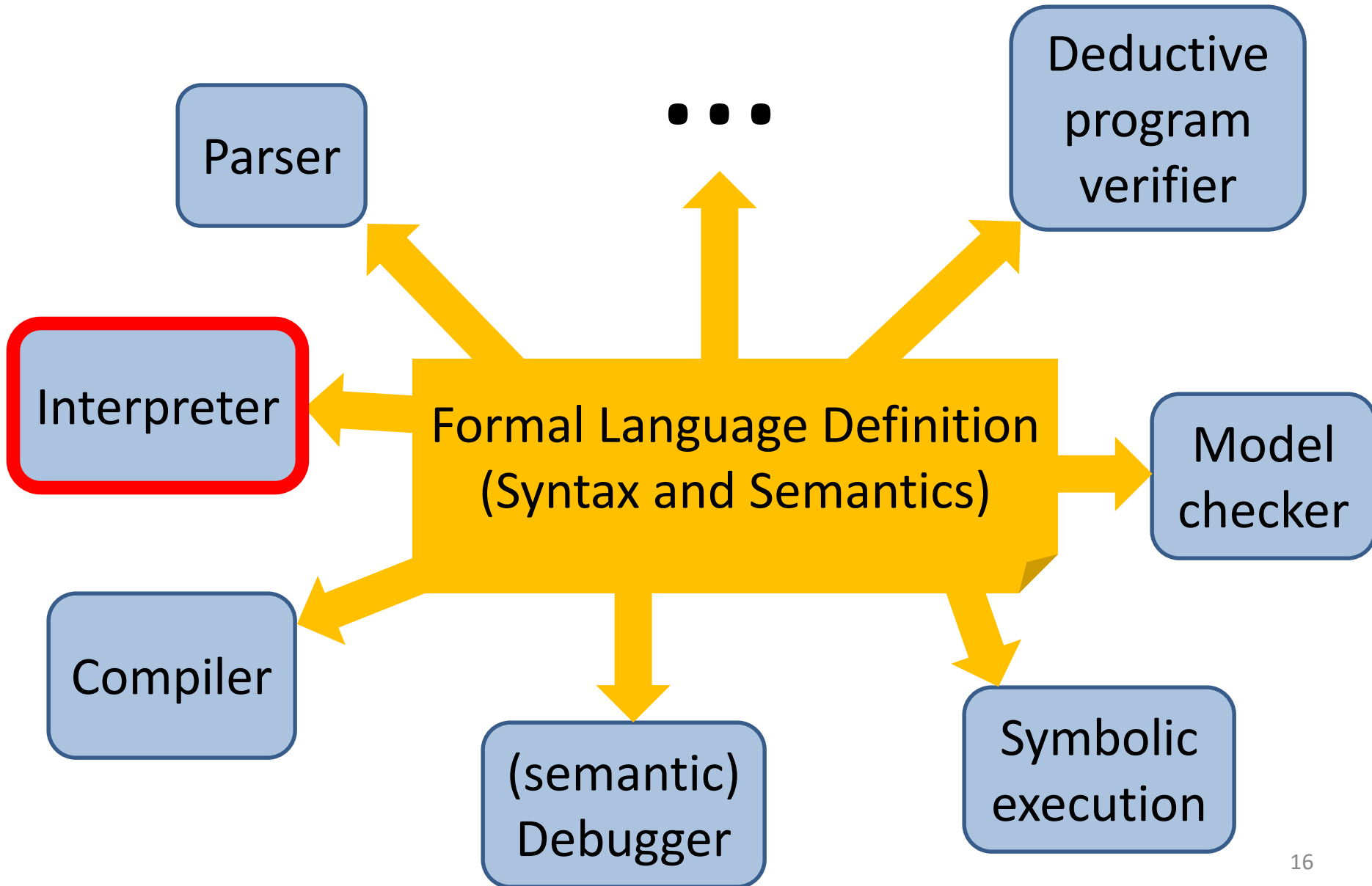
$\langle env \rangle \dots X | -> (_ \Rightarrow V) \dots \langle /env \rangle$

K Scales

Several large languages were recently defined in K:

- **JavaScript ES5**: by Park etal [PLDI'15]
 - Passes existing conformance test suite (2872 programs)
 - Found (confirmed) bugs in Chrome, IE, Firefox, Safari
- **Java 1.4**: by Bogdanas etal [POPL'15]
- **x86**: by Dasgupta etal [PLDI'19]
- **C11**: Ellison etal [POPL'12, PLDI'15]
 - 192 different types of undefined behavior
 - 10,000+ program tests (gcc torture tests, obfuscated C, ...)
 - Commercialized by startup (Runtime Verification, Inc.)
- + **EVM** [CSF'18], **Solidity**, **IELE**, **Plutus**, **Vyper**, ...

Ideal Language Framework Vision [K]



OCAML backend: K -> OCAML

1. Translate K lang def to OCAML
2. Compile OCAML code natively



**runtime
verification
match**

Code (6-int-overflow.c)

```
int main() {  
    short int a = 1;  
    int i;  
    for (i = 0; i < 15; i++) {  
        a *= 2;  
    }  
    return a;  
}
```

RV-Match: Commercial tool

- Instance of K -> OCAML with ISO C11 language
- an automatic debugger for subtle bugs [other tools can't find](#), with no false positives
- seamless integration with unit tests, build infrastructure, and continuous integration
- a platform for analyzing programs, boosting standards compliance and assurance

Conventional
compilers do not
detect problem

```
$ gcc 6-int-overflow.c  
$ ./a.out  
$  
$ kcc 6-int-overflow.c  
$ ./a.out
```

RV-Match's kcc tool precisely
detects and reports error, and
points to ISO C11 standard

```
Error: IMPL-CCV2
```

```
Description: Conversion to signed integer outside the range that can be represented.  
Type: Implementation defined behavior.  
See also: C11 sec. 6.3.1.3:3, J.3.5:1 item 4  
at main(6-int-overflow.c:29)
```

From RV-Match to Blockchain

- RV-Match currently commercialized within



BOEING

DENSO

Crafting the Core



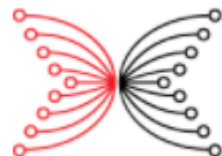
TOYOTA

INFO TECHNOLOGY
CENTER CO., LTD.

- The same technology, K, used for defining blockchain languages: EVM, eWASM, IELE, ...

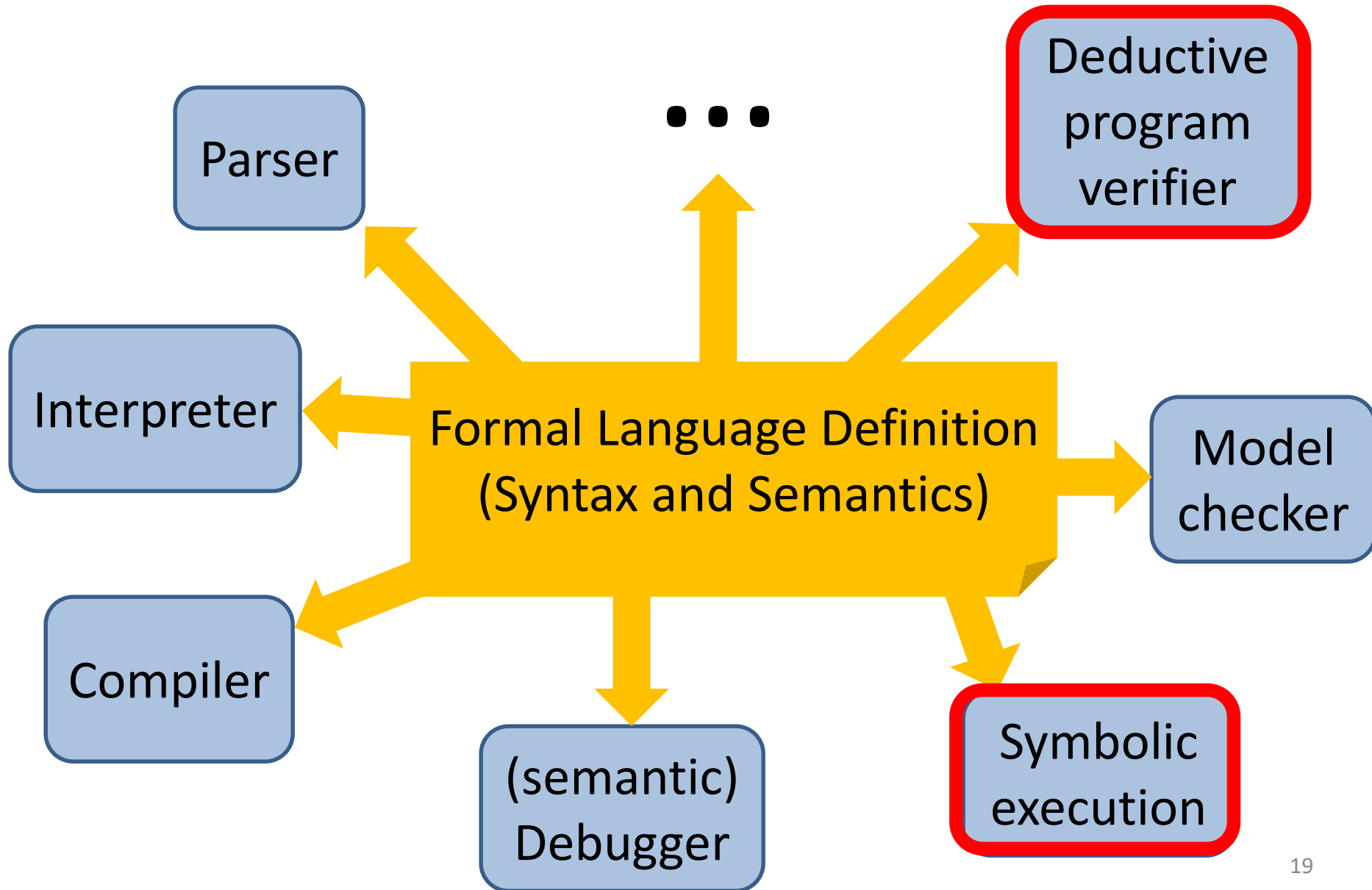


ethereum



INPUT | OUTPUT

Ideal Language Framework Vision [K]



State-of-the-Art

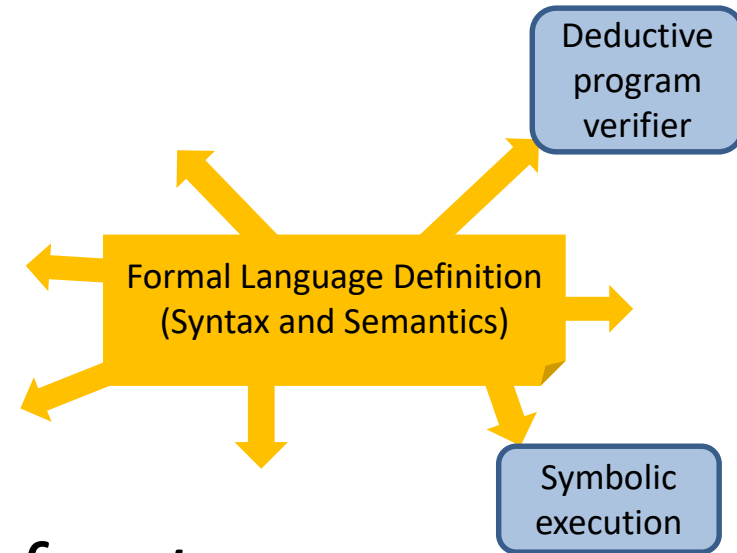
- **Redefine** the language using a **different** semantic approach (Hoare/separation/dynamic logic)
- **Language specific, non-executable, error-prone**

$$\frac{\mathcal{H} \vdash \{\psi \wedge e \neq 0\} s \{\psi\}}{\mathcal{H} \vdash \{\psi\} \text{while}(e) s \{\psi \wedge e = 0\}}$$

$$\frac{\mathcal{H} \cup \{\psi\} \text{proc}() \{\psi'\} \vdash \{\psi\} \text{body} \{\psi'\}}{\mathcal{H} \vdash \{\psi\} \text{proc}() \{\psi'\}}$$

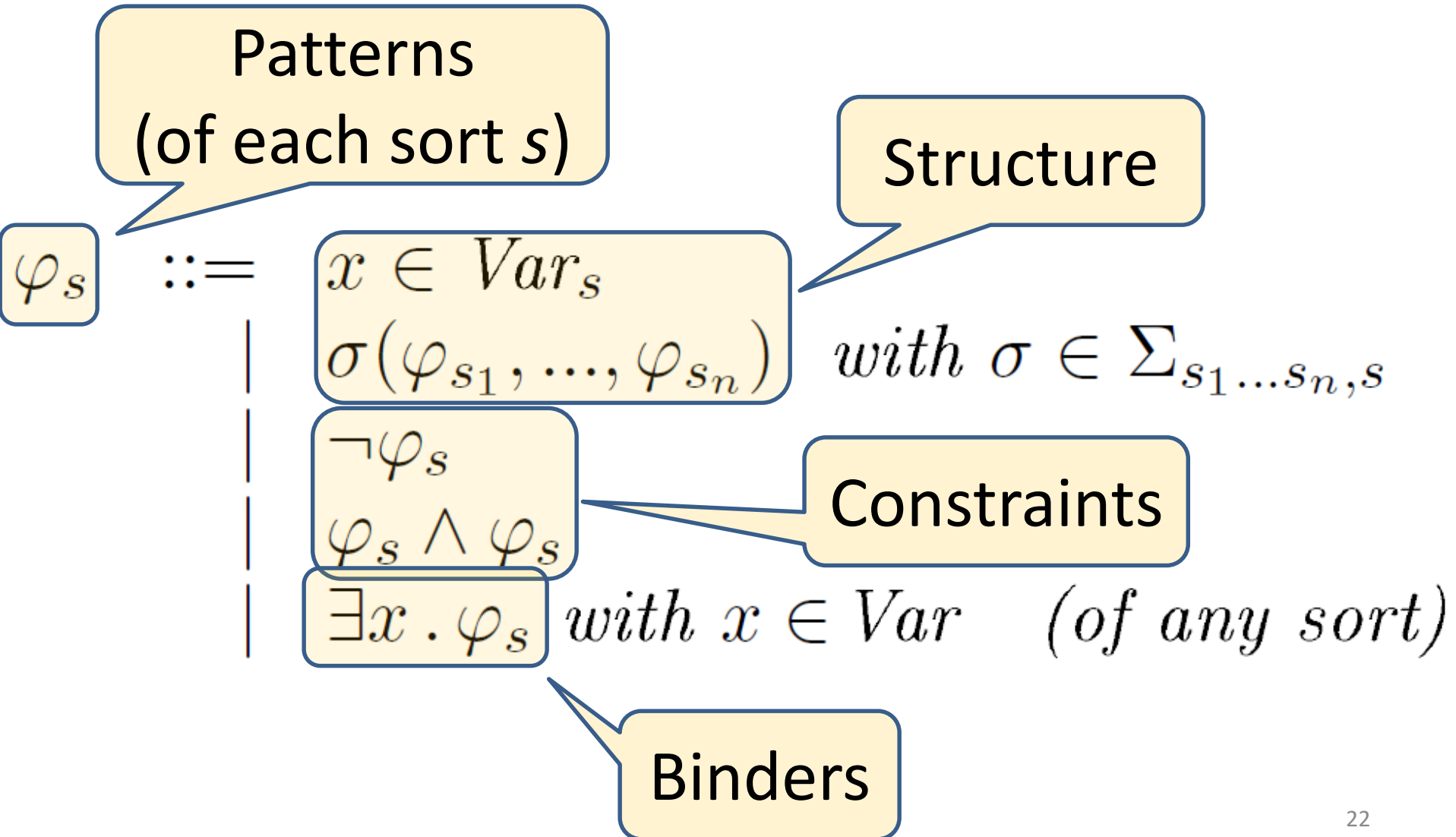
What We Want

- Use directly the trusted executable semantics!
- *Language-independent proof system*
 - Takes operational semantics as axioms
 - Derives reachability properties
 - Sound and relatively complete for all languages!



Matching Logic

[..., LICS'13, RTA'15, OOPSLA'16, FSCD'16, LMCS'17, ...]



Matching μ -Logic [LICS'19]

- Adding support for *recursion / induction*

$$\varphi_s ::= \dots \mid X:s \in \text{SVAR}_s \\ \mid \mu X:s.\varphi_s \quad \text{if } \varphi_s \text{ is positive in } X:s$$

$$(\text{PRE-FIXPOINT}) \quad \varphi[\mu X.\varphi/X] \rightarrow \mu X.\varphi$$

$$(\text{KNASTER-TARSKI}) \quad \frac{\varphi[\psi/X] \rightarrow \psi}{\mu X.\varphi \rightarrow \psi}$$

Expressiveness

- Important logics for program reasoning can be framed as matching logic theories / notations
 - First-order logic
 - Equality, membership, definedness, partial functions
 - Lambda / mu calculi (least/largest fixed points)
 - Modal logics
 - Hoare logics
 - Dynamic logics
 - LTL, CTL, CTL*
 - Separation logic
 - Reachability logic
 - ...

Reachability Logic (Semantics of K)

[LICS'13, RTA'14, RTA'15, OOPSLA'16]

- “Rewrite” rules over matching logic patterns:

$$\varphi \Rightarrow \varphi'$$

Can be expressed in matching logic:
 $\varphi \rightarrow \Diamond(\varphi')$ \Diamond is “weak eventually”

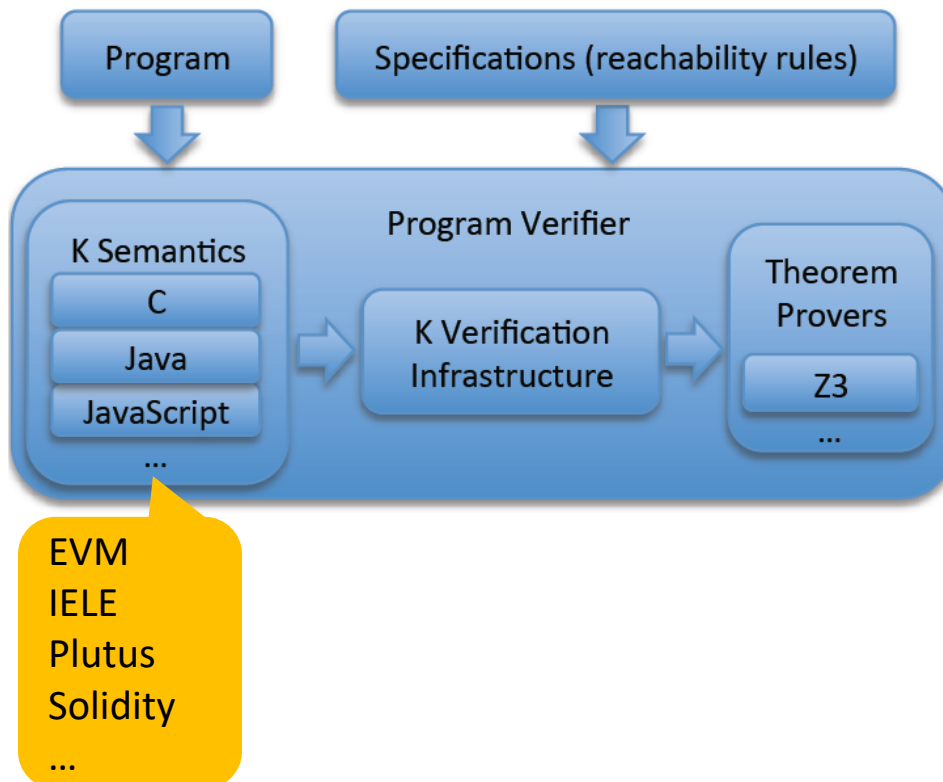
- Patterns generalize terms, so reachability rules capture rewriting, that is, operational semantics
- Reachability rules capture Hoare triples [FM'12]

$$\{Pre\} Code \{Post\} \equiv \widehat{Code} \wedge \widehat{Pre} \Rightarrow \epsilon \wedge \widehat{Post}$$

- Sound & relative complete proof system
 - Now proved as matching logic theorems

K = (Best Effort) Implementation of RL

- Reachability logic implemented in K, generically



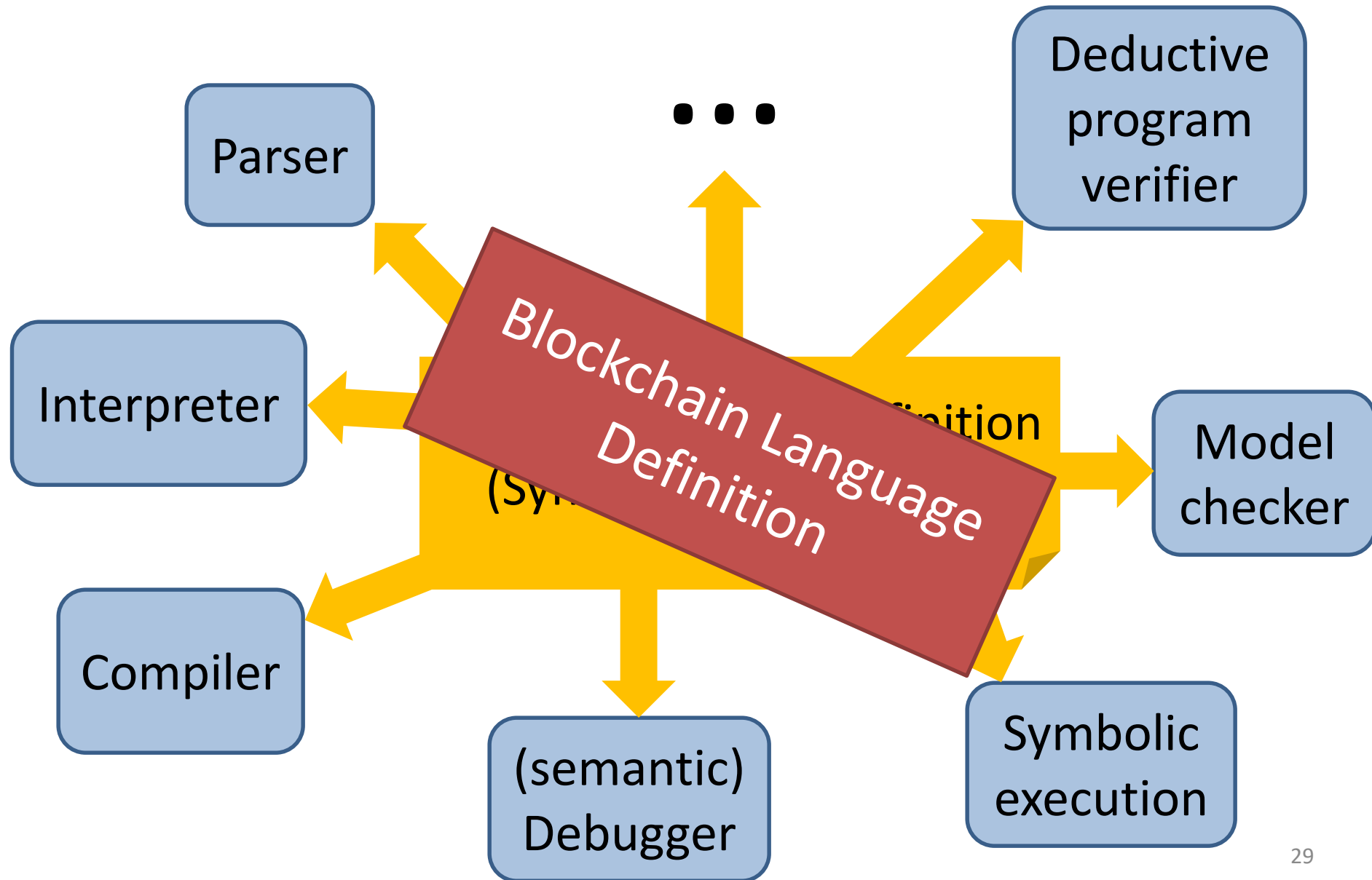
- Evaluated it with the existing semantics of C, Java, JavaScript, EVM, and several tricky programs
- Morale:
 - Performance is *comparable* with language-specific provers!

Sum $1+2+\dots+n$ in IMP: Main

```
rule
  <k>
    int n, sum;
    n = N:Int;
    sum = 0;
    while (!(n <= 0)) {
      sum = sum + n;
      n = n + -1;
    }
  =>
    .K
  </k>
  <state>
    .Map
  =>
    n    |-> 0
    sum  |-> ((N +Int 1) *Int N /Int 2)
  </state>
requires N >=Int 0
```

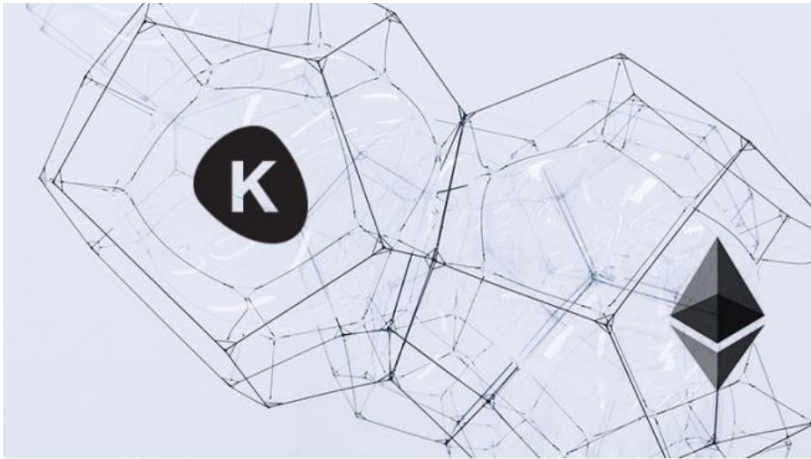
K for the Blockchain

K Framework Vision



KEVM: Semantics of the Ethereum Virtual Machine (EVM) in K

[CSL'18]



Complete semantics of EVM in K

- <https://github.com/kframework/evm-semantics>
- Passes 60,000+ tests of C++ reference implementation
- *8x (only!) slower than the C++ implementation*
- *Adoption by the Ethereum Foundation*

What Can We Do with KEVM?

1) *Formal documentation:* <http://jellopaper.org>

The screenshot shows a web browser displaying the JelloPaper website. The browser's address bar shows the URL <https://jellopaper.org/evm/>. The website has a blue header with the title "The EVM Jello" and a search bar. A sidebar menu on the left lists various topics, with "EVM Execution" selected. The main content area is titled "EVM Execution" and contains a section for "EVM Opcodes" and "EVM Control Flow".

KEVM: Semantics of EVM in K
Resources

- ⊞ EVM Execution
 - Overview
 - Configuration
 - Modal Semantics
 - ⊞ State Stacks
 - ⊞ Control Flow
 - ⊞ OpCode Execution
- ⊞ EVM Programs
- ⊞ Ethereum Gas Calculation
- ⊞ EVM Program Representations
- EVM Integration with Production Client
- Ethereum Simulations
- eDSL High-Level Notations

EVM Opcodes

EVM Control Flow

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

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

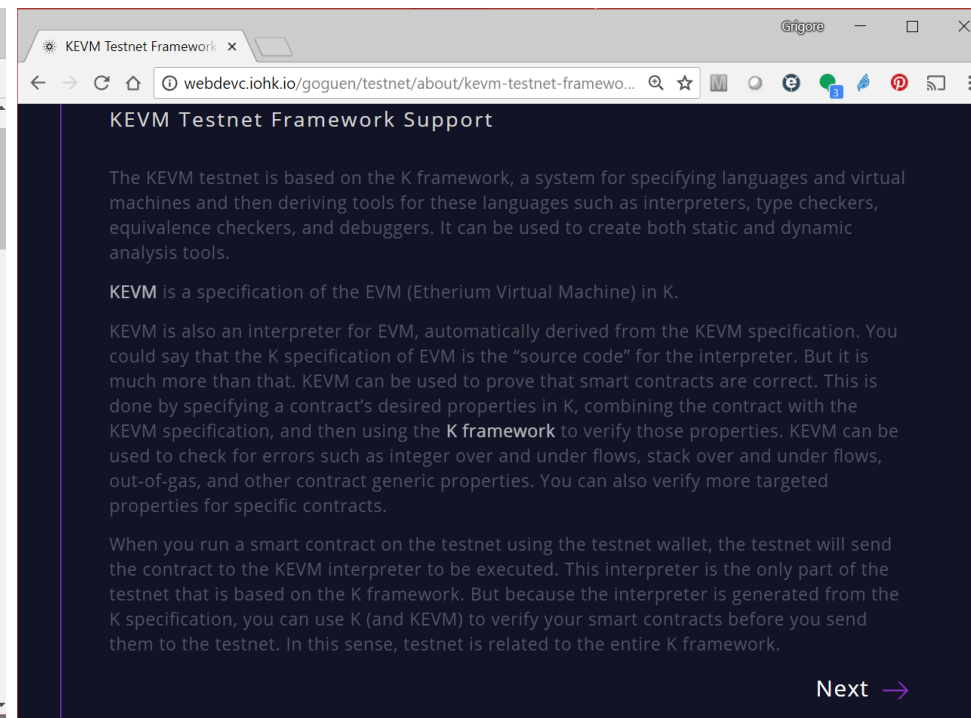
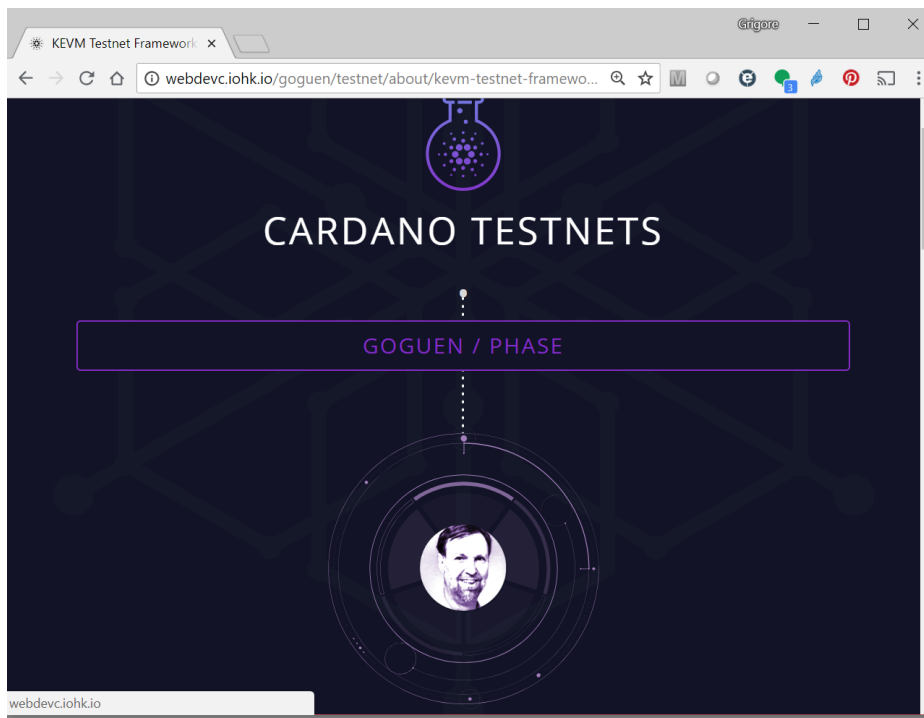
syntax UnStackOp ::= "JUMP"
// -----
rule <k> JUMP DEST => ... </k>
  <pc> _ => DEST </pc>
  <program> ... DEST |-> JUMPDEST ... </program>

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>
```

What Can We Do with KEVM?

2) *Generate and deploy correct-by-construction EVM client!* IOHK has just done that, in collaboration with RV, as a Cardano testnet:



What Can We Do with KEVM?

3) *Formally verify Ethereum smart contracts!* RV is doing that, commercially. RV also won first Ethereum Security grant to verify Casper.

The screenshot shows a web browser with two tabs. The active tab is titled "Announcing Beneficiaries of the Ethereum Foundation Grants" and displays a list of grant recipients. The "Runtime Verification" entry is highlighted with a red rectangle. The browser's address bar shows the URL "https://blog.ethereum.org/2018/03/07/announcing-beneficiaries-ethereum-foun...".

runtime verification

For

Comprehensive. End-to-end. Faithful

Comprehensive. We specify and verify smart contract owner the strongest

End-to-end. We start with the high-machine level. The last step is to ver

Faithful. We communicate with the captures the intended functionality. soon generate correctness certificate

Awardee List

Announcing Beneficiaries of the Ethereum Foundation Grants

- [L4 Research](#) – Scalability Grant – \$1.5M. State channels research.
- [Runtime Verification](#)** – Security Grant – \$500K. Casper contract formal verification.
- [ETHGlobal](#) – DevEx Grant* – \$200K. World-class developer conferences for Ethereum
- [Prysmatic Labs](#) – Scalability Grant – \$100K. Sharding implementation.
- [DDA](#) – #build Grant** – \$100K. Tokenless decentralized derivatives network + state channels R&D
- [Barcelona Supercomputing Center](#) – Scalability Grant – \$50K. Sharding simulation.
- [Plasma Taiwan Dev](#) – Scalability Grant – \$25K. Plasma implementation.
- [Ethers.js](#) – DevEx Grant – \$25K. Web3.js alternative.
- [Turbo Geth](#) – Scalability Grant – \$25K. Geth optimization.
- [Solium](#) – DevEx Grant – \$10K. Solidity static analyzer.
- [Alex Komarov](#) – Design Grant – \$10K. Key management UX study
- [\(Anonymous\)](#) – Hacktiership – \$10K. Deterministic WebAssembly.

[FSE'18]

Formalizing ERC20, ERC777, ... in K

- *K is very expressive*: can define not only languages, but also *token specifications and protocols*
- To formally verify smart contracts, we also formalized token specifications, multisigs, etc.:
 - [ERC20](#), [ERC777](#), [many others](#)
- All our specs are *language-independent*!
 - i.e., not specific to Solidity, not even to EVM
- We had the *first verified ERC20 contracts*!
 - Written both in Solidity and in Vyper
- Others use or integrate our framework and specs:
 - Consensys, DappHub ([KLab](#)), ETHWorks ([Waffle](#)), Gnosis



Chris Shields says:

December 7, 2017 at 7:44 pm | Edit

This is the coolest thing I've seen since the invention of smart contracts!

Smart Contract Verification Workflow

Transfers `_value` amount of tokens to address `_to`, and MUST fire the `Transfer` event. The function SHOULD `throw` if the `_from` account balance does not have enough tokens to spend.

Note Transfers of 0 values MUST be treated as normal transfers and fire the `Transfer` event.

```
function transfer(address _to, uint256 _value) returns (bool success)
```

1

ERC20 Informal
Business Logic

2

ERC20-K
formal
executable
high-level spec

Formalize

Refine

3

ERC20-EVM
formal executable
low-level spec
that contains all
EVM details

```
rule
  transfer(T, V) => true
  caller: F
  account:
    id: F
    balance: BF => BF - V
  account:
    id: T
    balance: BT => BT + V
  log: . => Transfer(F,T,V)
requires
  0 <= V /\
  V <= BF /\
  BT + V <= MAXVALUE
```

```
[transfer]
callData: #abiCallData("transfer", #address(TO_ID),
#uint256(VALUE))
gas: {GASCAP} => _
refund: _ => _
requires:
  andBool 0 <=Int TO_ID   andBool TO_ID   <Int
(2 ^Int 160)
  andBool 0 <=Int VALUE   andBool VALUE
<Int (2 ^Int 256)
  andBool 0 <=Int BAL_FROM andBool
BAL_FROM <Int (2 ^Int 256)
  andBool 0 <=Int BAL_TO   andBool BAL_TO
<Int (2 ^Int 256)
```

```
[transfer-success]
k: #execute => (RETURN RET_ADDR:Int 32 ~> _)
localMem: .Map => ( .Map[ RET_ADDR :=
#asByteStackInWidth(1, 32) ] _:Map )
log: _:List ( .List =>
ListItem(#abiEventLog(ACCT_ID, "Transfer",
#indexed(#address(CALLER_ID
```

.....

.....

.....

Designing New (and Better) Blockchain Languages Using K

EVM Not Human Readable (among other nuisances)

If it must be
low-level, then
I prefer this:

```
define public @sum(%n) {  
    %result = 0  
condition:  
    %cond = cmp le %n, 0  
    br %cond, after_loop  
    %result = add %result, %n  
    %n = sub %n, 1  
    br condition  
after_loop:  
    ret %result  
}
```



```
PUSH(1, 0) ; PUSH  
; PUSH(1, 10) ; PUSH  
; JUMPDEST  
; PUSH(1, 0) ; PUSH  
; ISZERO ; PUSH(1,  
; PUSH(1, 32) ; MLOA  
; PUSH(1, 1)  
; PUSH(1, 10) ; JUM  
; JUMPDEST  
; PUSH(1, 0) ; MLOA
```

```
PUSH(1, 0) ; MSTORE  
PUSH(1, 32) ; MSTORE
```



A New Virtual Machine (and Language) for the Blockchain

- Incorporates learnings from defining KEVM and from using it to verify smart contracts
- Register-based machine, like LLVM; unbounded*
- *IELE was designed and implemented using formal methods and semantics from scratch!*
- Until IELE, only existing or toy languages have been given formal semantics in K
 - Not as exciting as designing new languages
 - We should use semantics as an intrinsic, active language design principle, not post-mortem

K Semantics of Other Blockchain Languages

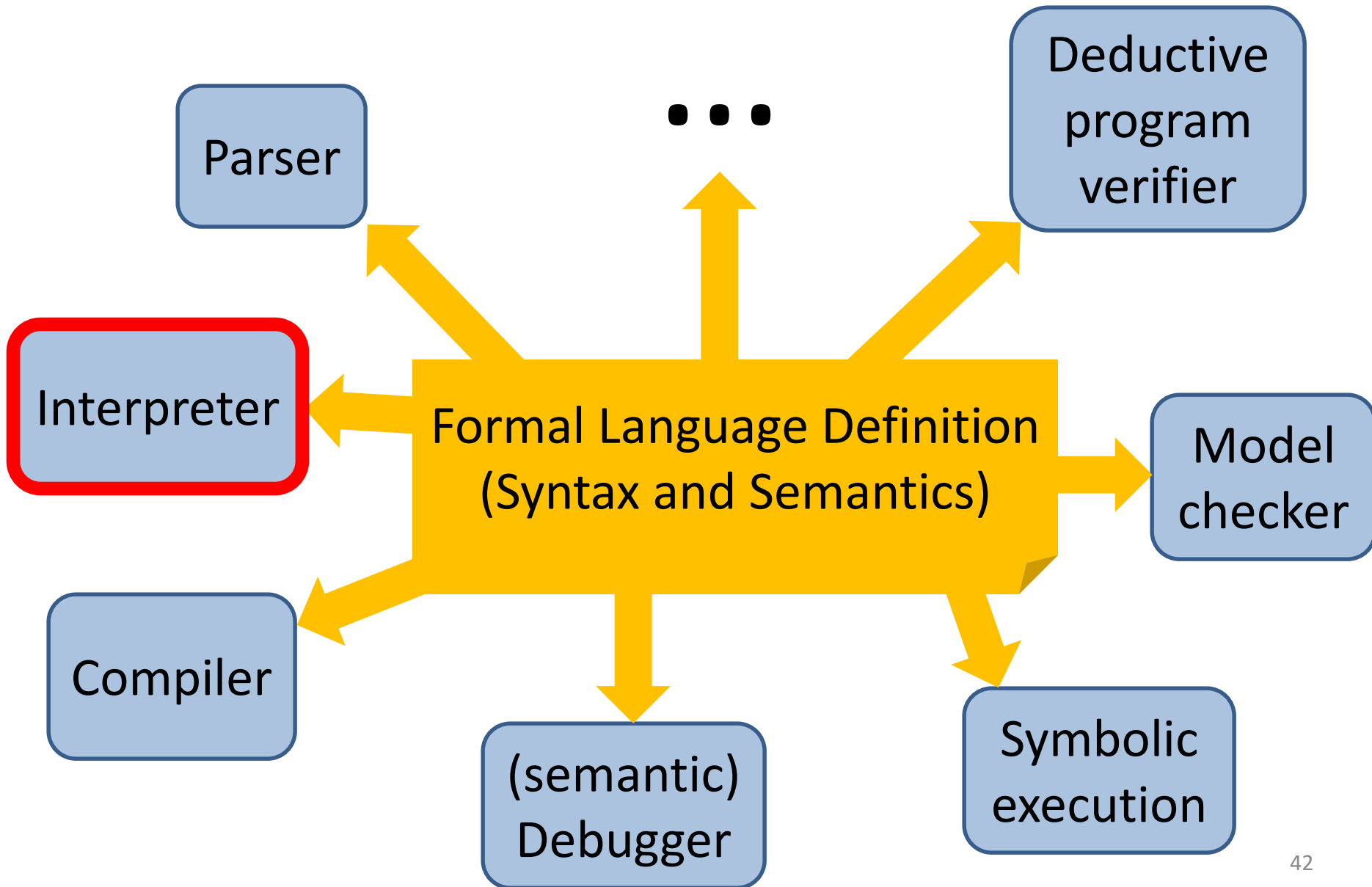
- **WASM** (web assembly) – in progress, in collaboration with the Ethereum Foundation
- **Solidity** – in progress, collaboration between RV and Sun Jun's group in Singapore
- **Plutus** (functional) – in progress, by RV following Phil Wadler's (@IOHK) design of the language
- **Vyper** – in progress, by RV in collaboration with the Ethereum Foundation
- ...

K Modelling and Verification of Blockchain Protocols

- K and rewriting can also be used to formally specify and verify consensus protocols, random number generators, etc.; same tool eco-system
- Done or ongoing:
 - Casper FFG (Ethereum Foundation)
 - RANDAO (Ethereum Foundation)
 - Casper CBC (Coordination Technology)
 - Serenity / ETH 2.0 (Ethereum Foundation)
- Several others planned or in discussions

Ongoing K Infrastructure Projects

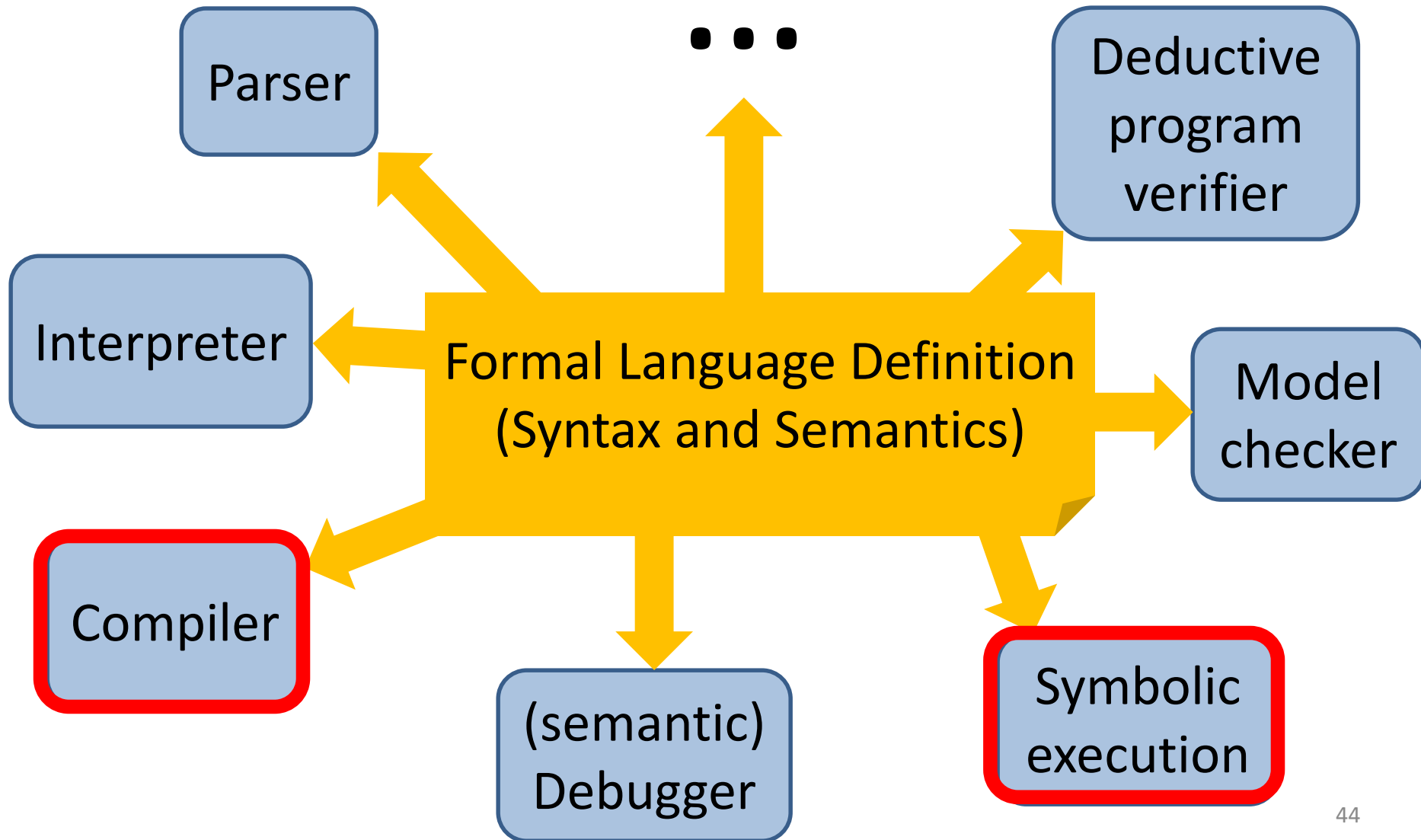
1. Fast LLVM (and IELE) Backend for K



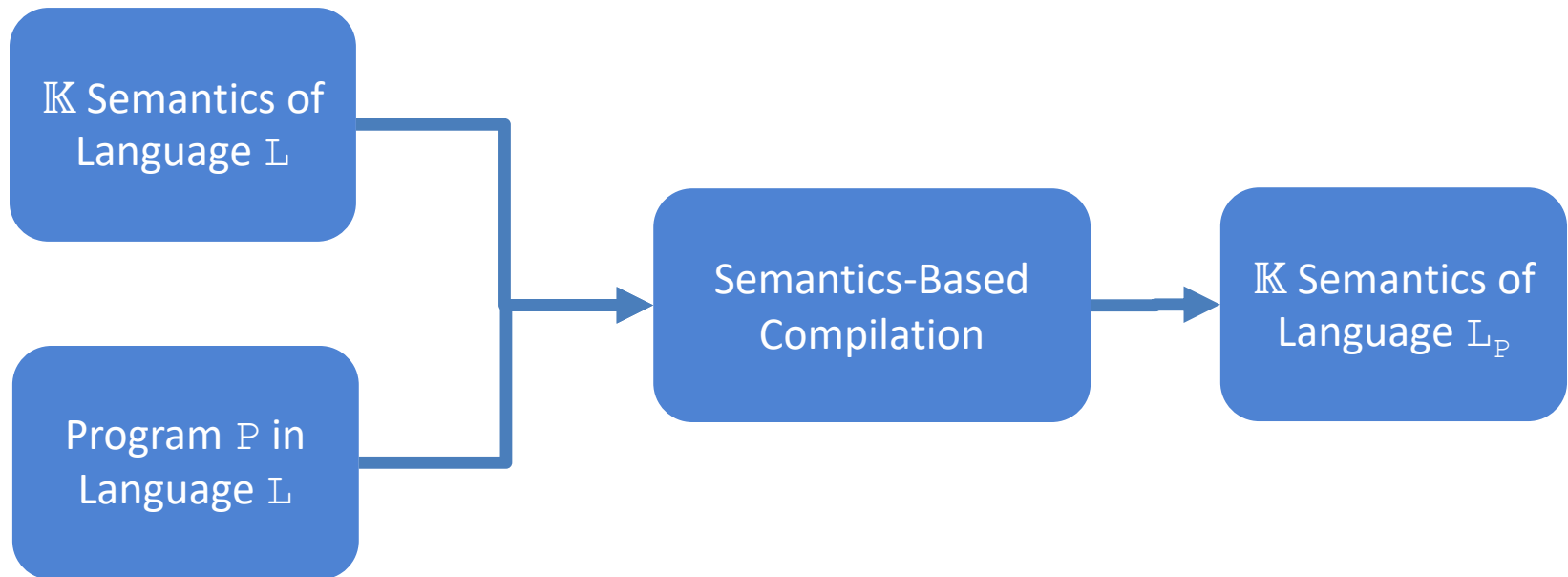
Fast LLVM (and IELE) Backend for K

- Current OCAML backend of K:
 - Fast enough to power RV-Match product and the KEVM and IELE VMs in testnets
 - But still one or two orders of magnitude slower than hand-crafted interpreters
- LLVM backend for K under development:
 - Take advantage of LLVM's optimizations / pipeline
 - Expected to compete with hand-written interpreters!
 - Will make language designers ask themselves the question
“Why would I implement an interpreter/VM by hand, when I can generate one automatically, correct-by-construction?”

2. Semantics-Based Compilation



Semantics-Based Compilation (SBC)



Goals

- Execution of P in \mathbb{L} equivalent to executing \mathbb{L}_P in a start configuration
- \mathbb{L}_P should be “as simple as possible”, only capturing exactly the dynamics of \mathbb{L} necessary to execute program P

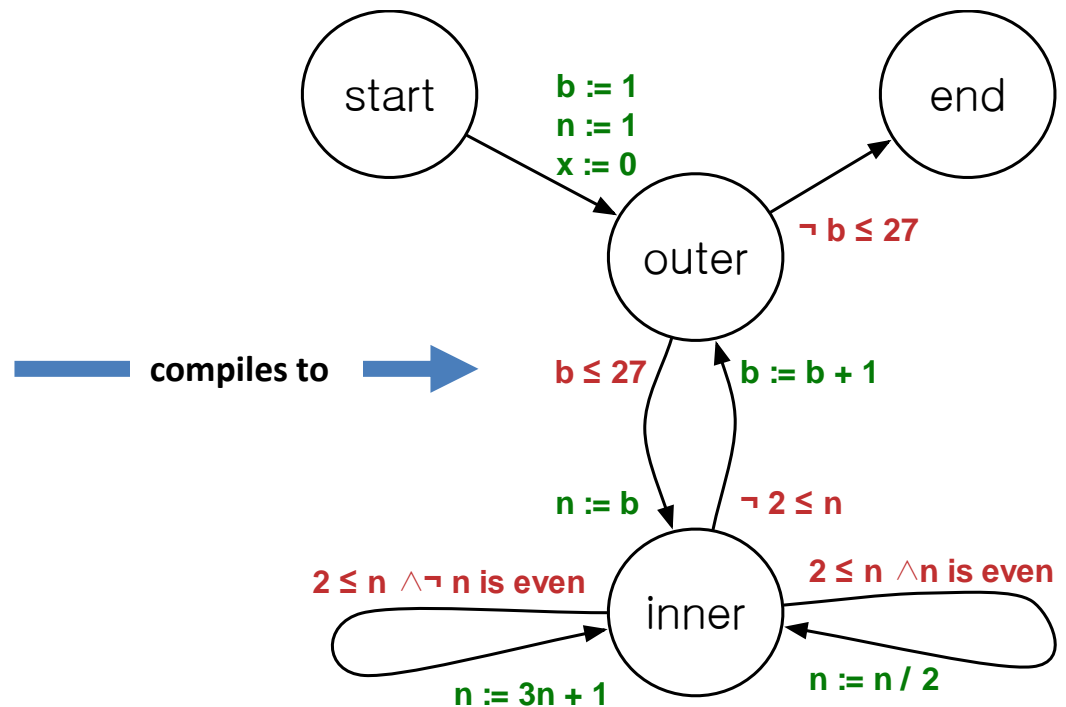
Semantics-Based Compilation (SBC)

Experiments with Early Prototype

```
// start
int b , n , x ;
b = 1 ; n = 1 ; x = 0 ;

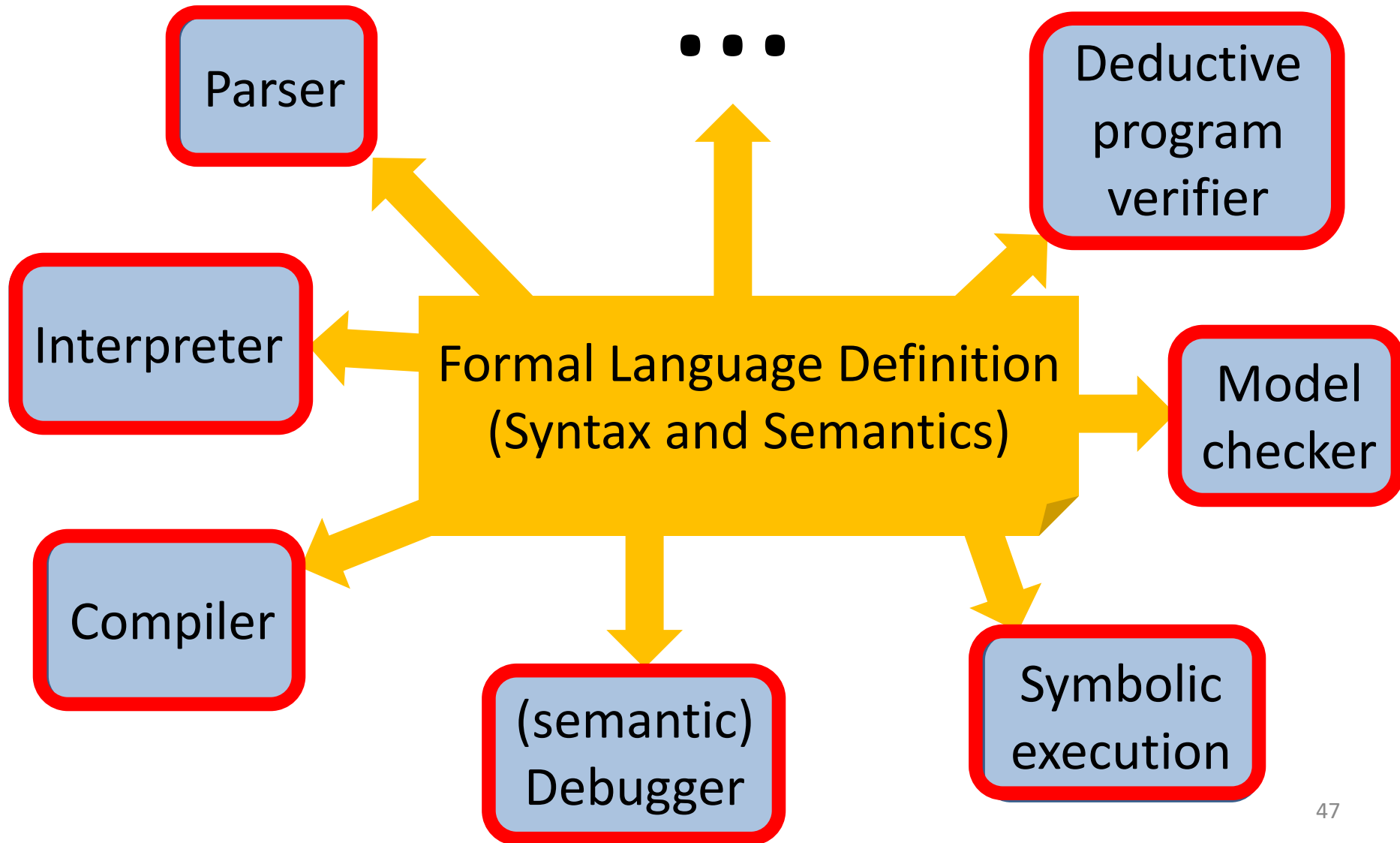
// outer
while (b <= 27) {
  n = b ;

  // inner
  while (2 <= n) {
    if (n <= ((n / 2) * 2))
    {
      n = n / 2 ;
    } else {
      n = (3 * n) + 1 ;
    }
    x = x + 1 ;
  }
  b = b + 1 ;
}
// end
```



Program	Original (s)	Compiled (s)	Speedup
sum.imp	70.6	7.3	9.7
collatz.imp	34.5	2.7	12.8
collatz-all.imp	77.4	5.7	13.6
krazy-loop.imp	67.6	3.3	20.5

3. Proof Object Generation



Proof Object Generation

- Each of the K tools is a best-effort implementation of proof search in Matching μ -Logic:

\mathcal{H}	(PROPOSITION ₁)	$\varphi_1 \rightarrow (\varphi_2 \rightarrow \varphi_1)$
	(PROPOSITION ₂)	$(\varphi_1 \rightarrow (\varphi_2 \rightarrow \varphi_3)) \rightarrow (\varphi_1 \rightarrow \varphi_2) \rightarrow (\varphi_1 \rightarrow \varphi_3)$
	(PROPOSITION ₃)	$(\neg\varphi_1 \rightarrow \neg\varphi_2) \rightarrow (\varphi_2 \rightarrow \varphi_1)$
		$\frac{\varphi_1 \quad \varphi_1 \rightarrow \varphi_2}{\varphi_2}$
	(MODUS PONENS)	$\frac{\varphi_2}{\varphi_2}$
	(VARIABLE SUBSTITUTION)	$\forall x. \varphi \rightarrow \varphi[y/x]$
	(\forall)	$\frac{\varphi}{\forall x. (\varphi_1 \rightarrow \varphi_2) \rightarrow (\varphi_1 \rightarrow \forall x. \varphi_2) \quad \text{if } x \notin FV(\varphi_1)}$
	(UNIVERSAL GENERALIZATION)	$\frac{\varphi}{\forall x. \varphi}$
	(PROPAGATION _{\perp})	$C_\sigma[\perp] \rightarrow \perp$
	(PROPAGATION _{\vee})	$C_\sigma[\varphi_1 \vee \varphi_2] \rightarrow C_\sigma[\varphi_1] \vee C_\sigma[\varphi_2]$
	(PROPAGATION _{\exists})	$C_\sigma[\exists x. \varphi] \rightarrow \exists x. C_\sigma[\varphi] \quad \text{if } x \notin FV(C_\sigma[\exists x. \varphi])$
		$\frac{\varphi_1 \rightarrow \varphi_2}{C_\sigma[\varphi_1] \rightarrow C_\sigma[\varphi_2]}$
	(FRAMING)	$C_\sigma[\varphi_1] \rightarrow C_\sigma[\varphi_2]$
	(EXISTENCE)	$\exists x. x$
	(SINGLETON VARIABLE)	$\neg(C_1[x \wedge \varphi] \wedge C_2[x \wedge \neg\varphi])$ where C_1 and C_2 are nested symbol contexts.
		$\frac{\varphi}{\varphi[\psi/X]}$
	(SET VARIABLE SUBSTITUTION)	$\varphi[\psi/X]$
	(PRE-FIXPOINT)	$\frac{\varphi[\mu X. \varphi/X] \rightarrow \mu X. \varphi}{\varphi[\psi/X] \rightarrow \psi}$
	(KNASTER-TARSKI)	$\mu X. \varphi \rightarrow \psi$

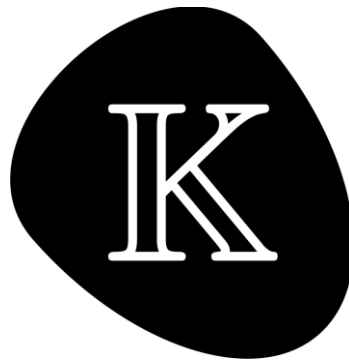
16 proof rules only!
Simple proof checker (200 LOC)!
In contrast, Coq has about 45 proof rules, and its proof checker has 8000+ lines of OCAML

- New Haskell backend of K will explicitly generate *proof objects* for verification tasks

Proof Object Generation

- No need to trust the (complex) K implementation, nor any company (including Runtime Verification)
 - It is known that program verifiers / tools can have bugs in spite of best efforts, bug finders and company prestige
- Proof objects act as *3rd-party checkable correctness certificates* on the blockchain, in a *proof-carrying code* style (proofs can be stored offchain, or snarked)
- In combination with *domain-specific languages* for requirements specifications, this will offer the highest level of software assurance known to man

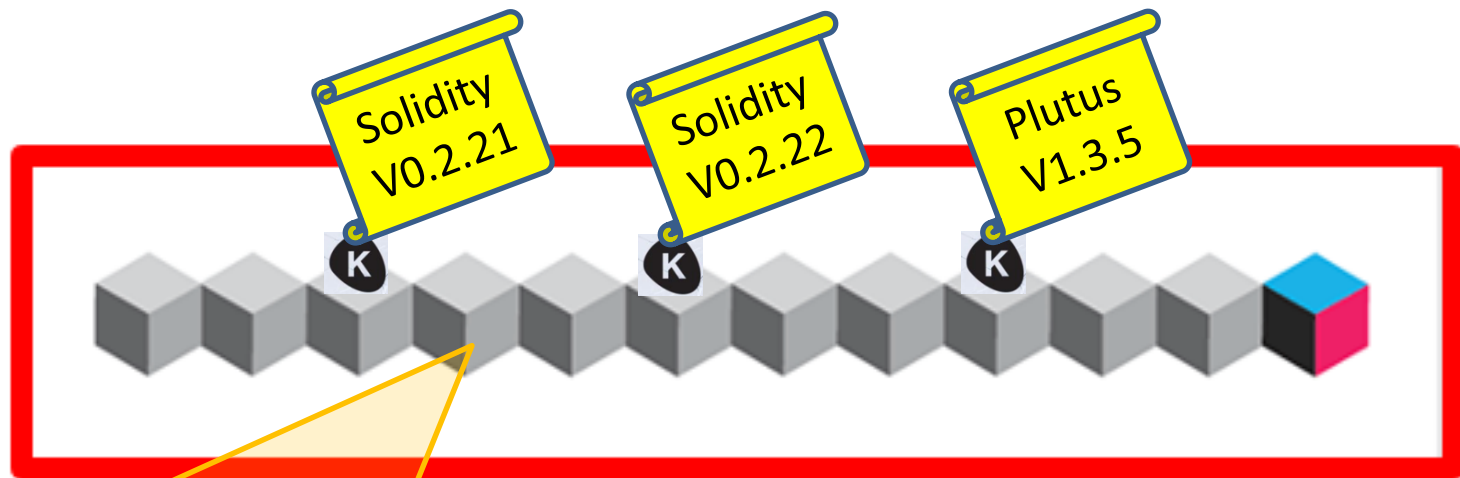
Ultimate Goal



a Universal Blockchain Technology

K – A Universal Blockchain Language

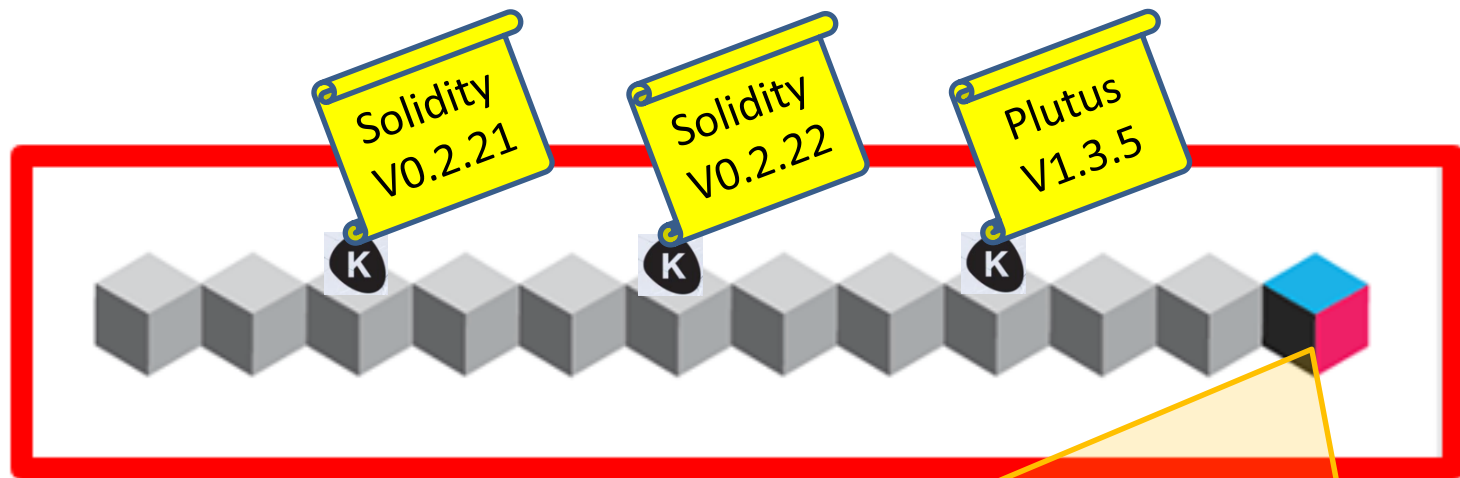
- We want to be able to write (provably correct) smart contracts in *any* programming language.
- All you need is a *K-powered blockchain!*



K language semantics will be stored on blockchain. Fast LLVM backend of K as execution engine / VM.

K – A Universal Blockchain Language

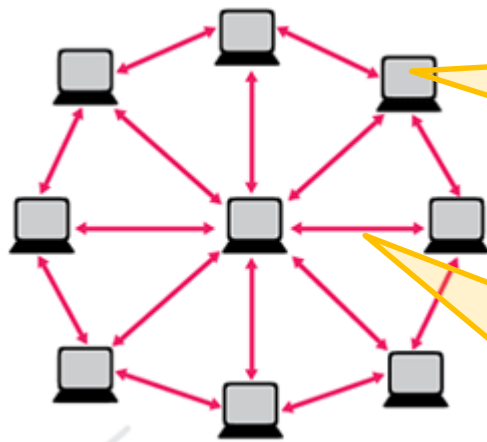
- *K-powered blockchain* enables (provably correct) smart contracts in *any* programming language!



1. Write contract P in any language, say L (unique address)
2. $SBC[L]$ your P into L_p ; verify P (or L_p) with K prover

K-Powered Blockchains

- K may be used one day to generate correct-by-construction (CBC) blockchains; not a dream, no!



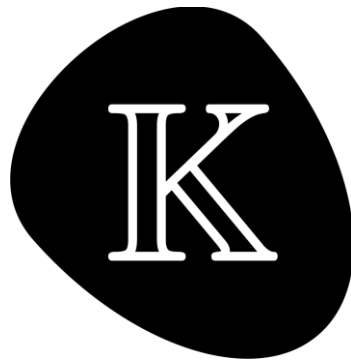
Each node is a K VM client (LLVM backend)

Consensus protocol formalized and verified in K, implementation generated from specification, CBC

K – A Universal Blockchain Language

- When all the projected K tools will be completed, K will provide everything we need to
 - Design new smart contract languages, add them in the blockchain and start using them right away, with auto-generated correct(!) implementations and tools
 - Same for virtual machines(!) and consensus protocols(!)
- Everything will be either a trusted formal specification or generated automatically from one
- Proof objects will serve as correctness certificates
- *Perfect. No compromise!*

Moreover...



a Ultimate Smart Contract Language

K as a Smart Contract Language

- Smart contracts implement transactions
 - Often using poorly designed and thus insecure languages, compilers and interpreters / VMs

K also implements transactions, directly!

- Indeed, each K rule instance *is* a transaction

- Each smart contract (Solidity, EVM, ...) requires a formal specification in order to be verified

K formal specifications are already executable!

- And indeed, they are validated by heavy testing

Hm, then why not write my smart contracts *directly* and *only* as K executable specifications?



Example: ERC20 Token in Solidity

- Snippet -

```
1  pragma solidity ^0.5.0;
2
3  import "../IERC20.sol";
4  import "../math/SafeMath.sol";
5
6  contract ERC20 is IERC20 {
7      using SafeMath for uint256;
8
9      mapping (address => uint256) private _balances;
10
11     function transfer(address to, uint256 value) public returns (bool) {
12         _transfer(msg.sender, to, value);
13         return true;
14     }
15
16     function _transfer(address from, address to, uint256 value) internal {
17         require(to != address(0), "ERC20: transfer to the zero address");
18
19         _balances[from] = _balances[from].sub(value);
20         _balances[to] = _balances[to].add(value);
21         emit Transfer(from, to, value);
22     }
23
24 }
```

Example: ERC20 Compiled to EVM

- Snippet -

Opcodes:

```
PUSH1 0x80 PUSH1 0x40 MSTORE CALLVALUE DUP1 ISZERO PUSH2 0x10 JUMPI PUSH1 0x0 DUP1 REVERT JUMPDEST POP PUSH2 0x423 DUP1
PUSH2 0x20 PUSH1 0x0 CODECOPY PUSH1 0x0 RETURN INVALID PUSH1 0x80 PUSH1 0x40 MSTORE CALLVALUE DUP1 ISZERO PUSH2 0x10
JUMPI PUSH1 0x0 DUP1 REVERT JUMPDEST POP PUSH1 0x4 CALLDATASIZE LT PUSH2 0x2B JUMPI PUSH1 0x0 CALLDATALOAD PUSH1 0xE0
SHR DUP1 PUSH4 0xA90F
60806 CALLDATASIZE SUB PUSH1 0x1 DUP1 6576
00080 CALLDATALOAD PUSH1 0x1 DUP1 1515
15815 CALLDATALOAD SWAP1 0 DUP1 6827
3ffff DUP3 ISZERO ISZERO 1 RETURN 5260
04018 JUMPDEST PUSH1 0x0 P JUMP ffff
fffff JUMPDEST PUSH1 0x0 ffff
fffff 0xFFFFFFFFFFFFFFFF ffff
fffff 0x8C379A0000000000 3fff
ff165 DUP3 DUP2 SUB DUP1 20 ADD ffff
19055 ADD SWAP2 POP POP H1 0x40 0208
952ba 0xFFFFFFFFFFFFFFFF USH20 8daa
00000 0x20 ADD SWAP1 DU RE PUSH1 0000
6e206 JUMP JUMPDEST PUS FFFF AND 696f
f08c3 0xFFFFFFFFFFFFFFFF H1 0x0 0517
61646 KECCAK256 DUP2 SW FFFFFFFF 3a20
46f20 AND PUSH20 0xFFFF x20 ADD 2207
PUSH1 0x0 KECCAK25 PUSH20
0xFFFFFFFFFFFFFFFF STORE PUSH1
0x20 ADD SWAP1 DUP2 MSTORE PUSH1 0x20 ADD PUSH1 0x0 KECCAK256 DUP2 SWAP1 MSTORE POP DUP2 PUSH20
0xFFFFFFFFFFFFFFFF AND DUP4 PUSH20 0xFFFFFFFFFFFFFFFF AND PUSH32
0xDDF252AD1BE2C89B69C2B068FC378DAA952BA7F163C4A11628F55A4DF523B3EF DUP4 PUSH1 0x40 MLOAD DUP1 DUP3 DUP2 MSTORE PUSH1
0x20 ADD SWAP2 POP POP PUSH1 0x40 MLOAD DUP1 SWAP2 SUB SWAP1 LOG3 POP POP POP JUMP JUMPDEST PUSH1 0x0 DUP3 DUP3 GT
```

- Unreadable
- Slow: ~25ms to execute (ganache)
- Untrusted compiler, so it needs to be formally verified to be trusted
 - We formally verify it using KEVM against the following K specification:

K Specification of ERC20

- Snippet, Sugared -

```
rule transfer(To, V) => true
  caller: From
  account: id: From balance: BalanceFrom => BalanceFrom - V
  account: id: To balance: BalanceTo => BalanceTo + V
  log: . => Transfer(From, To, V)
requires 0 <= V <= BalanceFrom /\ BalanceTo + V <= MAXVALUE
```

- Formal, yet understandable by non-experts
- Executable, thus testable (for increased confidence)
- Fast: ~2ms to execute with LLVM backend of K
- No compiler required
- Correct-by-construction, no code to formally verify
- *Use K as programming language for smart contracts!*

Conclusion: It Can Be Done!

