Formal Design, Implementation and Verification of Blockchain Languages

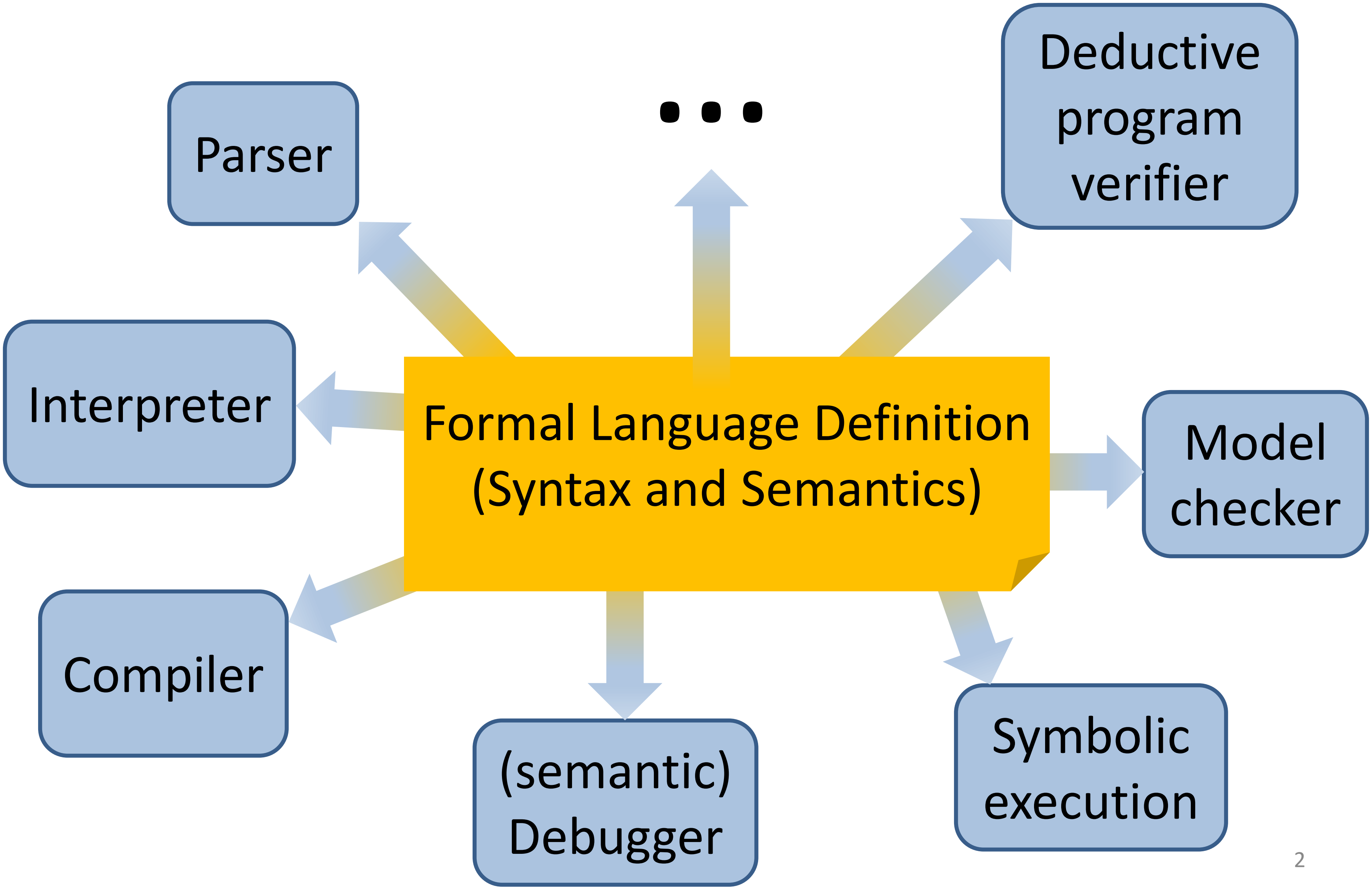
Grigore Rosu

University of Illinois at Urbana-Champaign

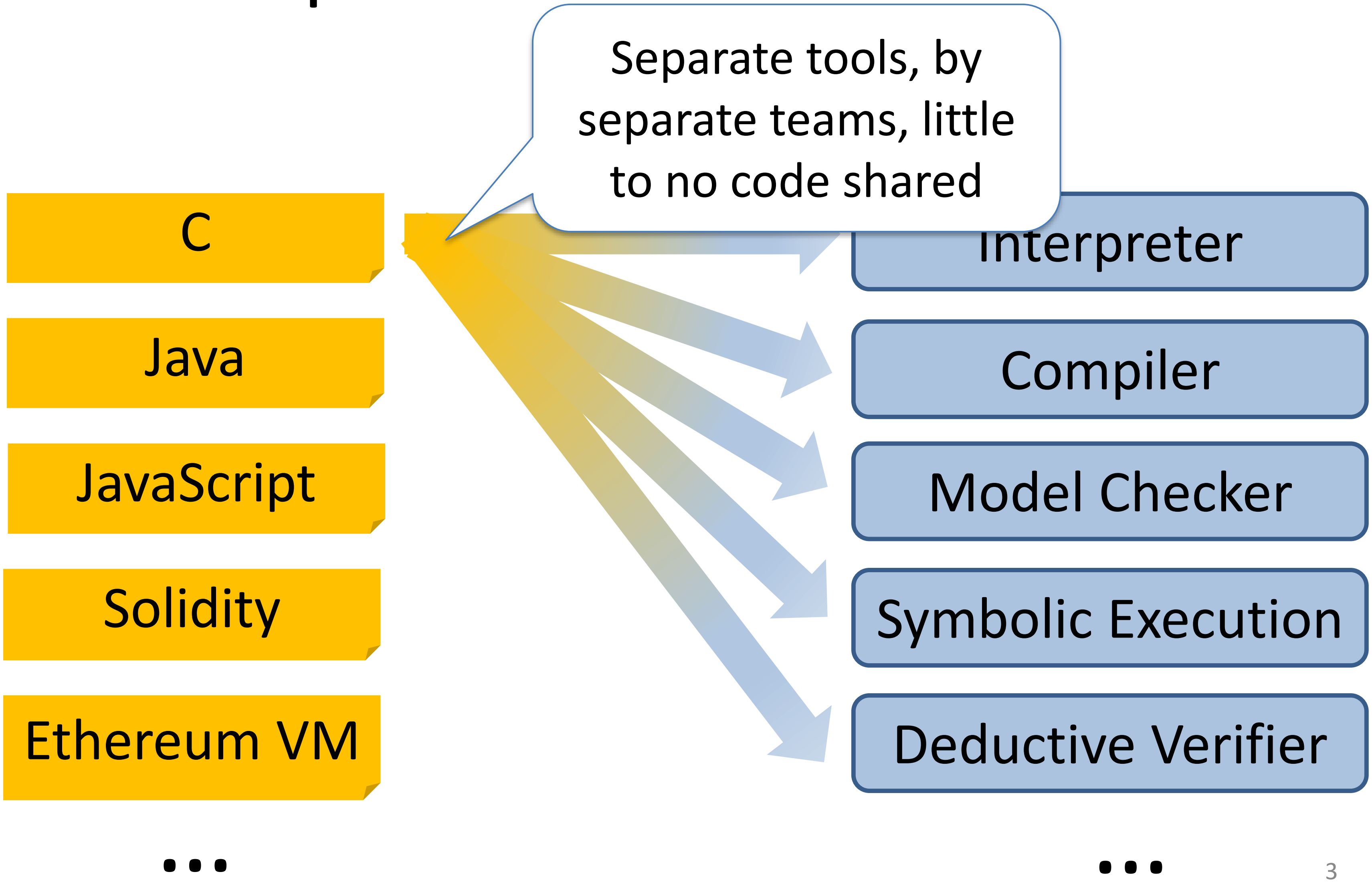
President & CEO, Runtime Verification Inc.

Waterloo, 2019-10-05

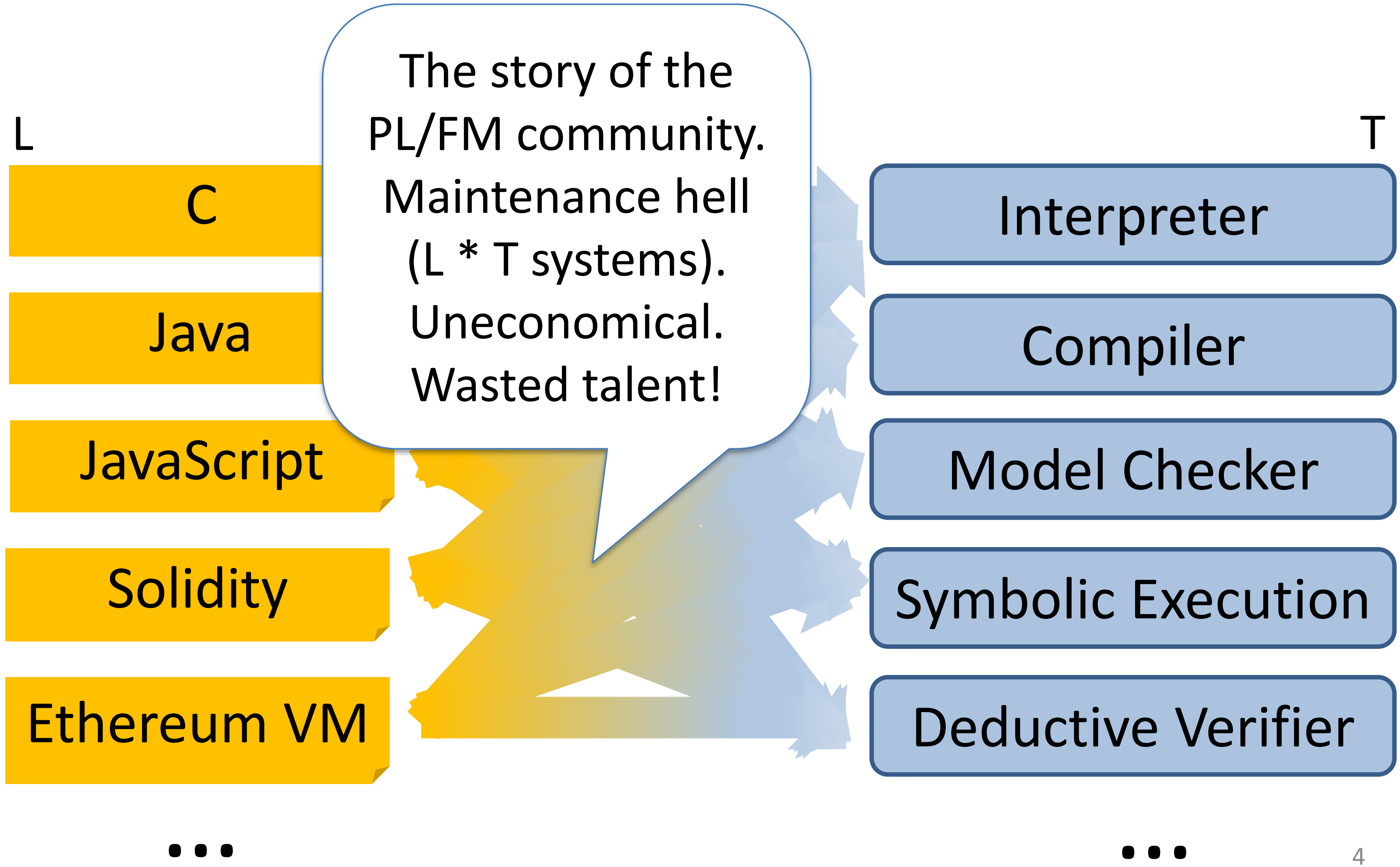
Ideal Language Framework Vision



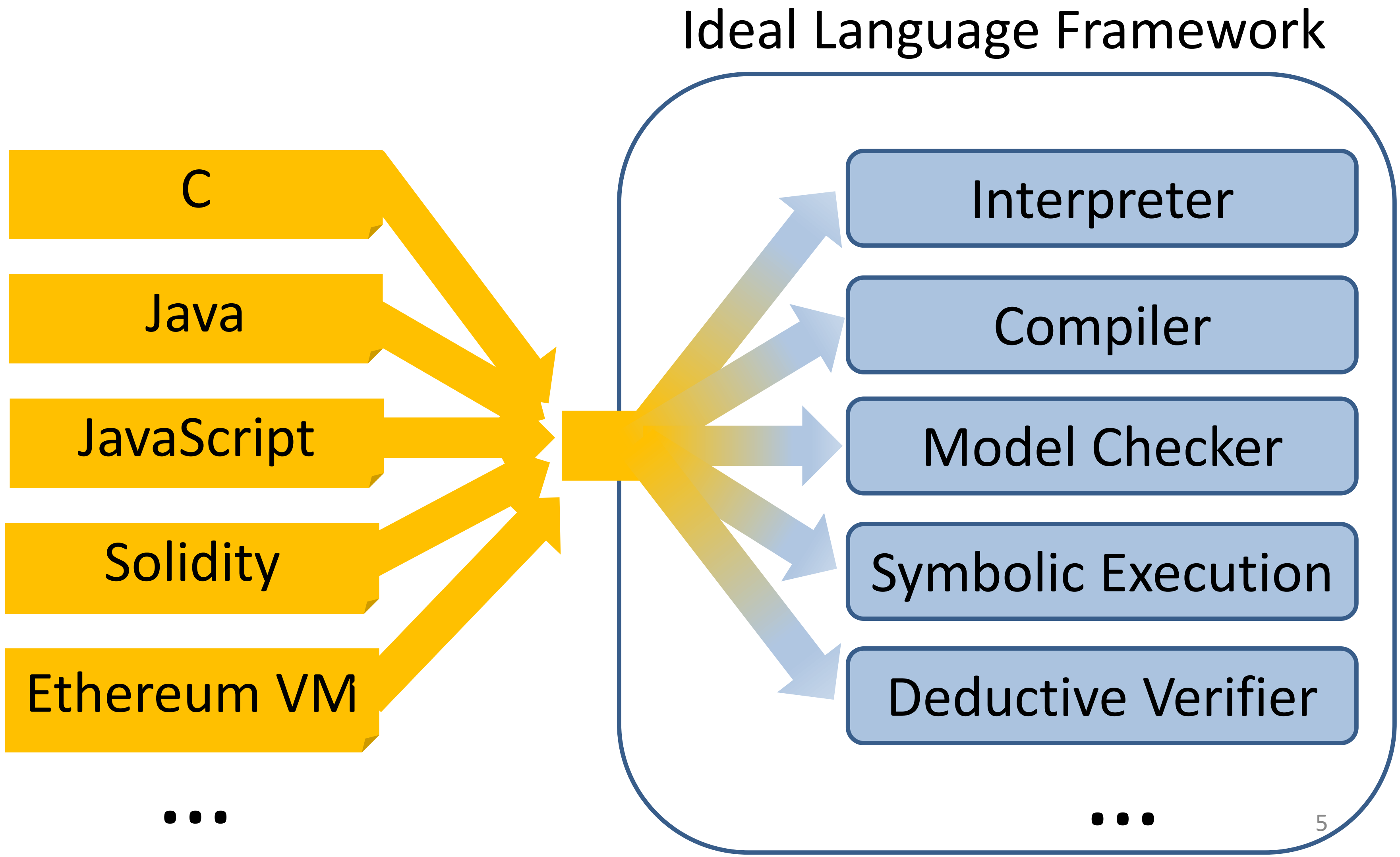
Current State-of-the-Art
- Sharp Contrast to Ideal Vision -



Current State-of-the-Art
- Sharp Contrast to Ideal Vision -



How It Should Be



Our Attempt: the K Framework
http://kframework.org

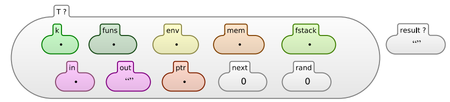
- We tried various semantic styles, for >15y and >100 top-tier conference and journal papers:
  - Small-/big-step SOS; Evaluation contexts; Abstract machines (CC, CK, CEK, SECD, …); Chemical abstract machine; Axiomatic; Continuations; Denotational;…
- But each of the above had limitations
  - Especially 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 )
               | scanf("%d", Exp ) [strict]
               | NULL
               | PointerId
               | (int*)malloc( Exp *sizeof(int)) [strict]
               | free( Exp ) [strict]
               | * Exp [strict]
               | Exp [ Exp ]
               | Exp = Exp [strict(2)]
               | Id ( List{Exp} ) [strict(2)]
               | Id ()
               | random()
               | srandom( Exp ) [strict]
  SYNTAX Stmt ::= Exp ; [strict]
                | {}
                | { StmtList }
                | if( Exp ) Stmt
                | if( Exp ) Stmt else Stmt [strict(1)]
                | 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
                        | List{Bottom}
  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
```

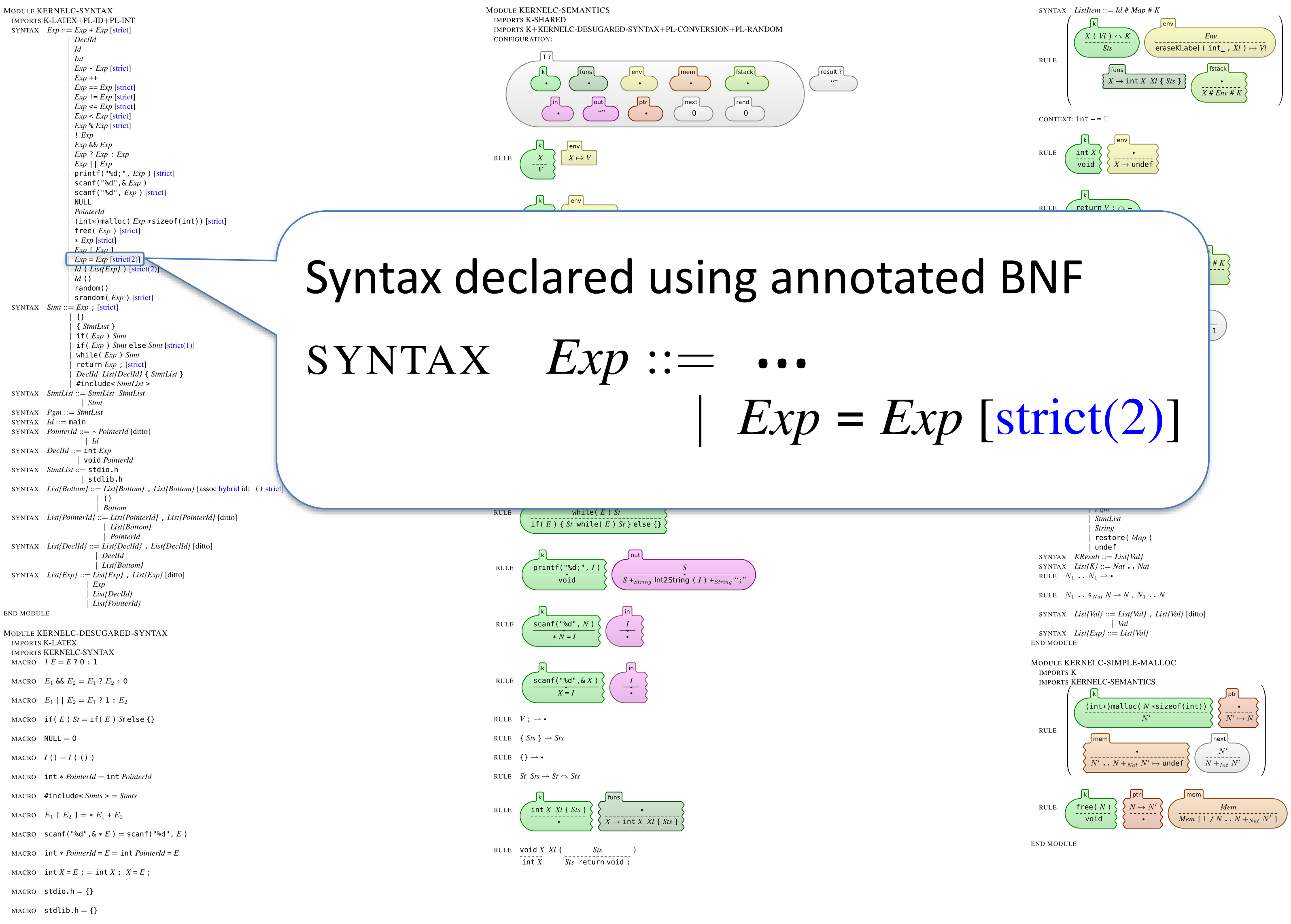
```
MODULE KERNELC-SEMANTICS
  IMPORTS K-SHARED
  IMPORTS K+KERNELC-DESUGARED-SYNTAX+PL-CONVERSION+PL-RANDOM
  CONFIGURATION:
```




```
  RULE <k> X => V ...</k> <env>... X |-> V ...</env>
  RULE <k> X ++ => I ...</k> <env>... X |-> (I => I +Int 1) ...</env>
  RULE <k> X = V => V ...</k> <env>... X |-> (_ => V) ...</env>
  RULE I1 + I2 => I1 +Int I2
  RULE I1 - I2 => I1 -Int I2
  RULE I1 % I2 => I1 %Int I2        when I2 !=Int 0
  RULE I1 <= I2 => Bool2Int ( I1 <=Int I2 )
  RULE I1 < I2 => Bool2Int ( I1 <Int I2 )
  RULE I1 == I2 => Bool2Int ( I1 ==Int I2 )
  RULE I1 != I2 => Bool2Int ( I1 !=Int I2 )
  RULE _?_:_ => if(_)_else_
  RULE if( I ) _ else St => St      when I ==Int 0
  RULE if( I ) St else _ => St      when ¬Bool I ==Int 0
  RULE <k> while( E ) St => if( E ) { St while( E ) St } else {} ...</k>
  RULE <k> printf("%d;", I ) => void ...</k> <out> S => S +String Int2String ( I ) +String “;” </out>
  RULE <k> scanf("%d", N ) => * N = I ...</k> <in> I => • ...</in>
  RULE <k> scanf("%d",& X ) => X = I ...</k> <in> I => • ...</in>
  RULE V ; => •
  RULE { Sts } => Sts
  RULE {} => •
  RULE St Sts => St ↷ Sts
  RULE <k> int X Xl { Sts } => • ...</k> <funs>... • => X |-> int X Xl { Sts } ...</funs>
  RULE (void X => int X) Xl { (Sts => Sts return void ;) }
  SYNTAX ListItem ::= Id # Map # K
  RULE ( <k> X ( Vl ) ↷ K => Sts </k> <env> Env => eraseKLabel ( int_ , Xl ) |-> Vl </env>
         <funs>... X |-> int X Xl { Sts } ...</funs> <fstack> • => X # Env # K ...</fstack> )
  CONTEXT: int _ = □
  RULE <k> int X => void ...</k> <env>... • => X |-> undef ...</env>
  RULE <k> return V ; ↷ _ => V </k>
  RULE <k> V ↷ (• => K) </k> <env> _ => Env </env> <fstack> _ # Env # K => • ...</fstack>
  RULE <k> random() => randomRandom ( N' ) ...</k> <rand> N' => N' +Nat 1 </rand>
  RULE <k> srandom( I ) => void ...</k>
  CONTEXT: * □ = _
  CONTEXT: * □ ++
  SYNTAX Val ::= Int
               | void
  SYNTAX Exp ::= Val
  SYNTAX K ::= List{DeclId}
             | List{Exp}
             | List{PointerId}
             | Pgm
             | StmtList
             | String
             | restore( Map )
             | undef
  SYNTAX KResult ::= List{Val}
  SYNTAX List{K} ::= Nat .. Nat
  RULE N1 .. N1 => •
  RULE N1 .. sNat N => N ,, N1 .. N
  SYNTAX List{Val} ::= List{Val} , List{Val} [ditto]
                     | Val
  SYNTAX List{Exp} ::= List{Val}
END MODULE
```

```
MODULE KERNELC-SIMPLE-MALLOC
  IMPORTS K
  IMPORTS KERNELC-SEMANTICS
  RULE ( <k> (int*)malloc( N *sizeof(int)) => N' ...</k> <ptr>... • => N' |-> N ...</ptr>
         <mem>... • => N' .. N +Nat N' |-> undef ...</mem> <next> N' => N +Int N' </next> )
  RULE <k> free( N ) => void ...</k> <ptr>... N |-> N' => • ...</ptr> <mem> Mem => Mem [⊥ / N .. N +Nat N' ] </mem>
END MODULE
```

Complete K Definition of KernelC



Syntax declared using annotated BNF

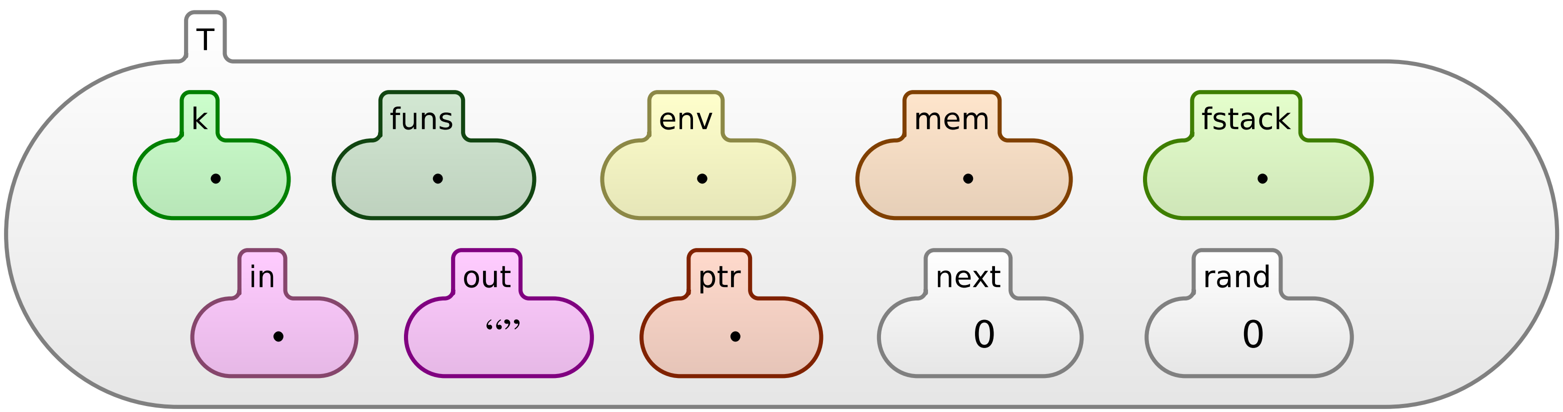
SYNTAX Exp ::= ...

| Exp = Exp [strict(2)]

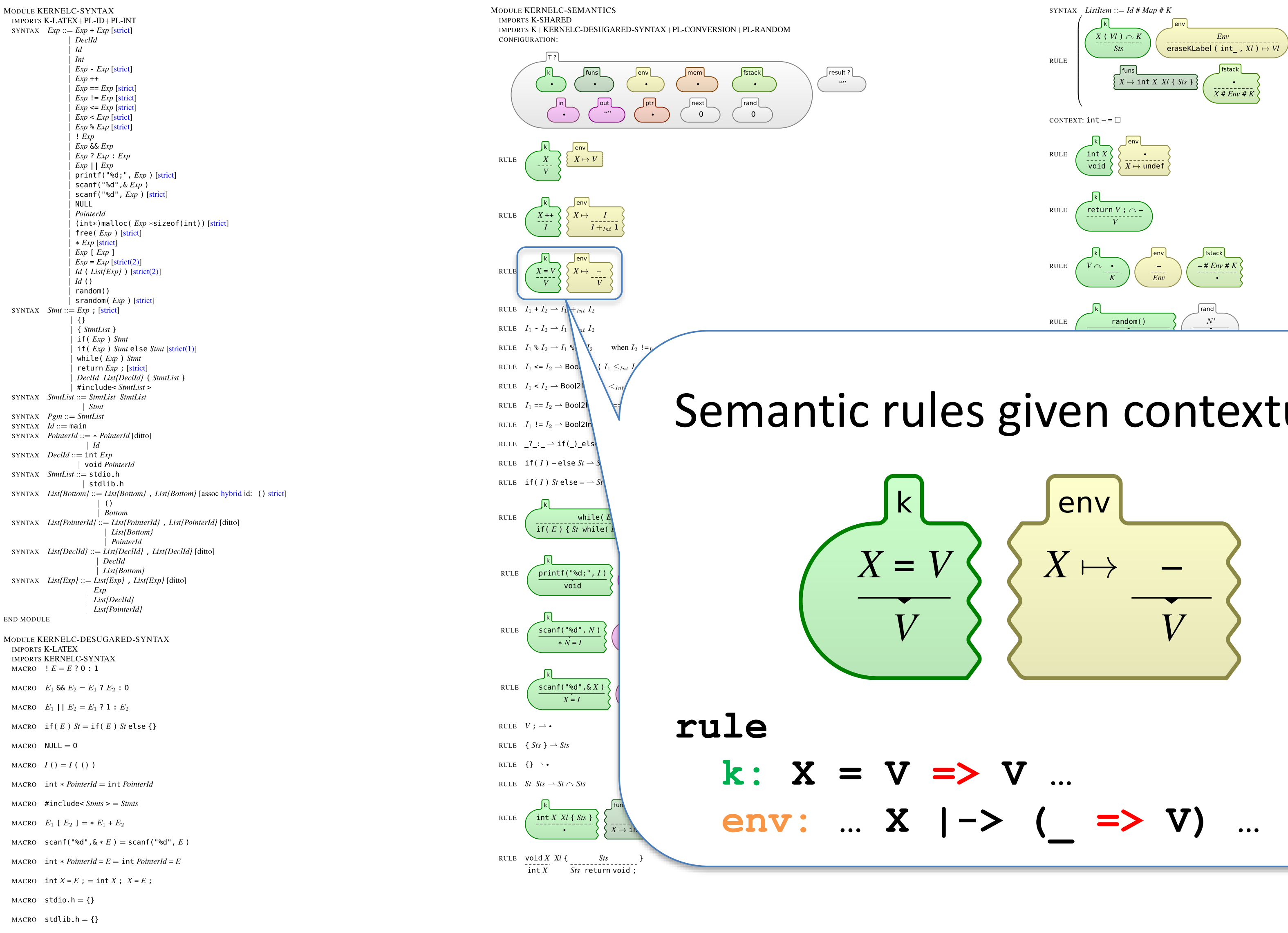
Complete K Definition of KernelC

Configuration given as a nested cell structure.

Leaves can be sets, multisets, lists, maps, or syntax



Complete K Definition of KernelC



Semantic rules given contextually

```
rule
  k: X = V => V ...
  env: ... X |-> (_ => V) ...
```

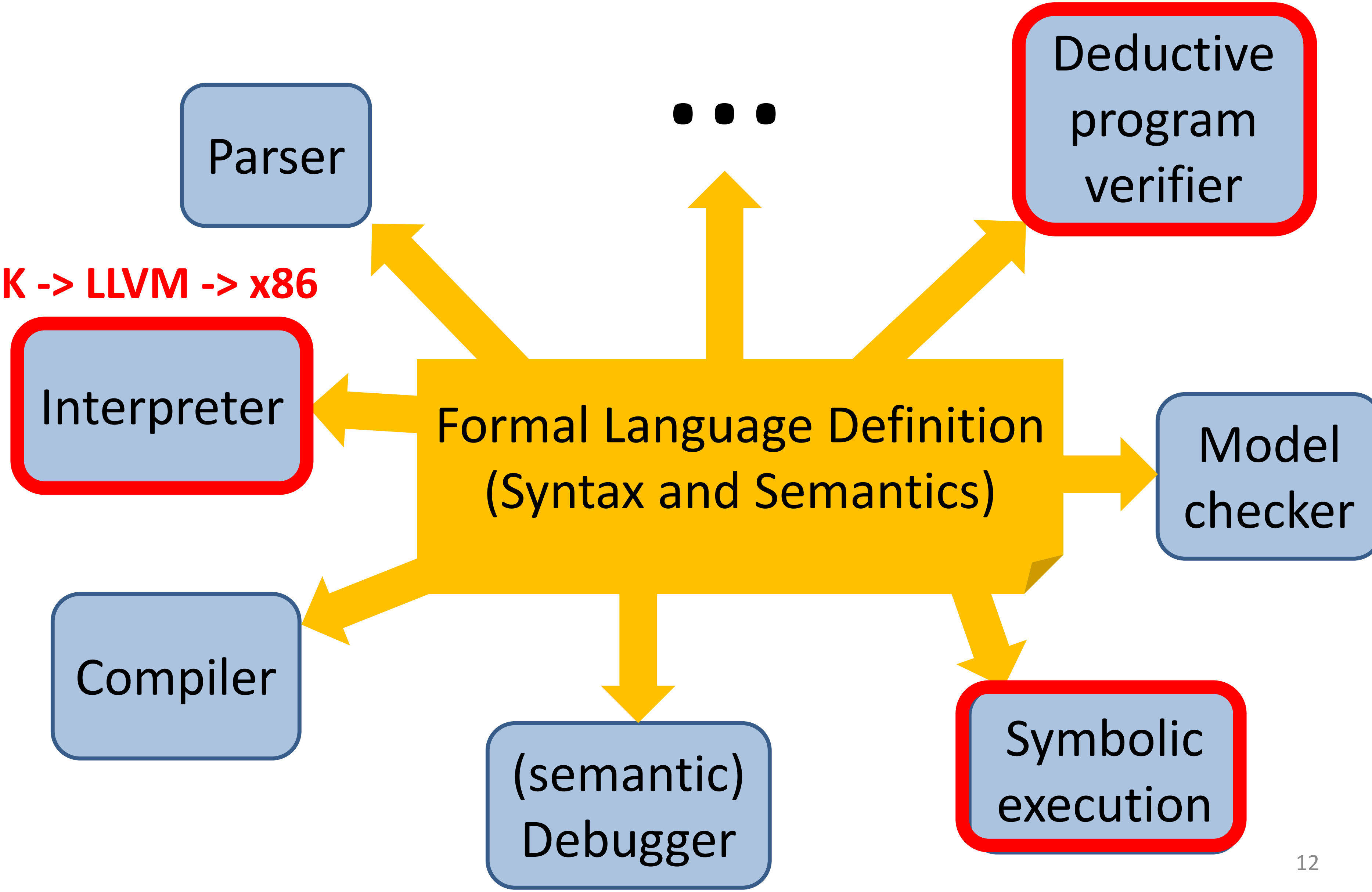
K Scales

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 [FM'19], Plutus, Vyper, …

Ideal Language Framework Vision [K]



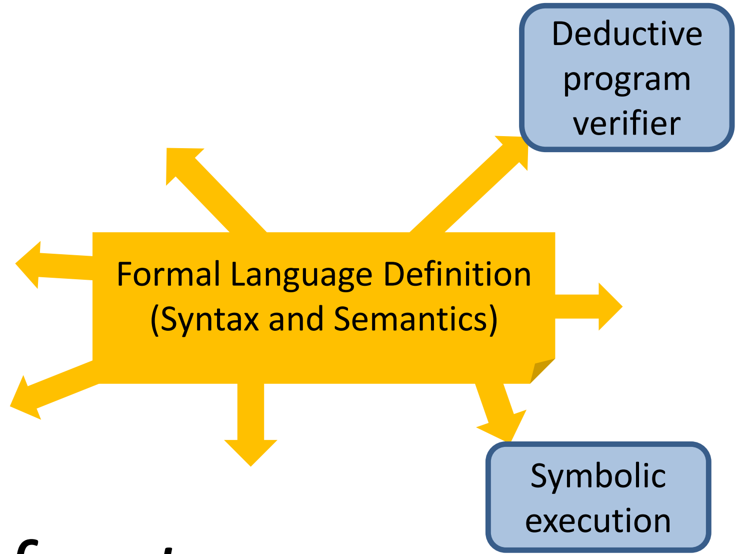
State-of-the-Art

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

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

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

Ideal Scenario



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

Matching μ-Logic

[..., LICS'13, RTA'15, OOPSLA'16, FSCD'16, LMCS'17, LICS'19]

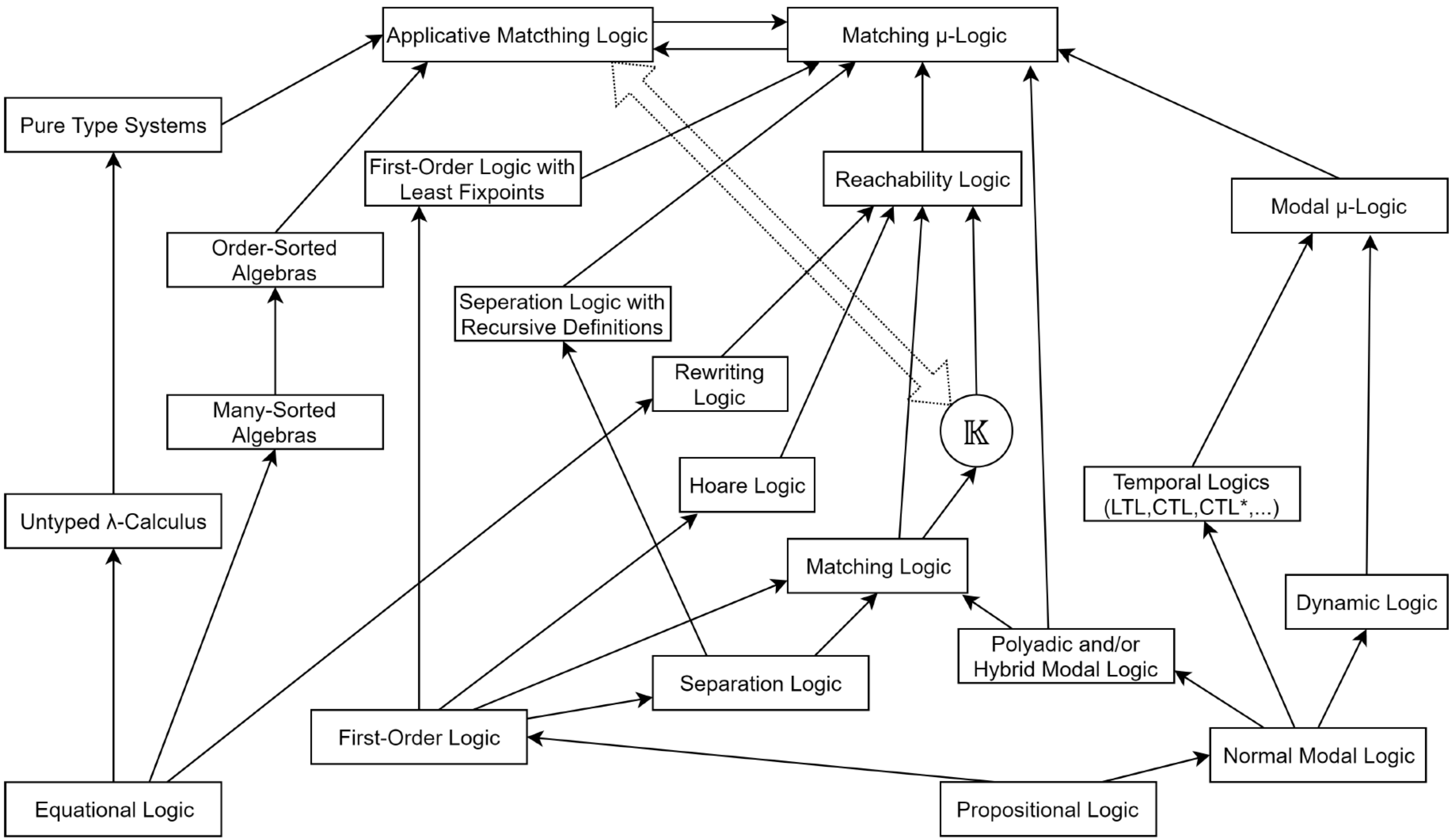
$\mathcal{H}_\mu$ = $\mathcal{H}$ + (Set Variable Substitution, Pre-Fixpoint, Knaster-Tarski)

(Proposition$_1$)	$\varphi_1 \to (\varphi_2 \to \varphi_1)$
(Proposition$_2$)	$(\varphi_1 \to (\varphi_2 \to \varphi_3)) \to (\varphi_1 \to \varphi_2) \to (\varphi_1 \to \varphi_3)$
(Proposition$_3$)	$(\neg\varphi_1 \to \neg\varphi_2) \to (\varphi_2 \to \varphi_1)$
(Modus Ponens)	$\dfrac{\varphi_1 \quad \varphi_1 \to \varphi_2}{\varphi_2}$
(Variable Substitution)	$\forall x.\varphi \to \varphi[y/x]$
($\forall$)	$\forall x.(\varphi_1 \to \varphi_2) \to (\varphi_1 \to \forall x.\varphi_2)$ if $x \notin FV(\varphi_1)$
(Universal Generalization)	$\dfrac{\varphi}{\forall x.\varphi}$
(Propagation$_\bot$)	$C_\sigma[\bot] \to \bot$
(Propagation$_\vee$)	$C_\sigma[\varphi_1 \vee \varphi_2] \to C_\sigma[\varphi_1] \vee C_\sigma[\varphi_2]$
(Propagation$_\exists$)	$C_\sigma[\exists x.\varphi] \to \exists x.C_\sigma[\varphi]$ if $x \notin FV(C_\sigma[\exists x.\varphi])$
(Framing)	$\dfrac{\varphi_1 \to \varphi_2}{C_\sigma[\varphi_1] \to C_\sigma[\varphi_2]}$
(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.
(Set Variable Substitution)	$\dfrac{\varphi}{\varphi[\psi/X]}$
(Pre-Fixpoint)	$\varphi[\mu X.\, \varphi/X] \to \mu X.\varphi$
(Knaster-Tarski)	$\dfrac{\varphi[\psi/X] \to \psi}{\mu X.\, \varphi \to \psi}$

Rules from (Proposition$_1$) to (Singleton Variable) form $\mathcal{H}$.

16 proof rules only!
Simple proof checker
(200 LOC, vs Coq's 8000)!

Expressiveness of Matching μ-Logic



Reachability Logic (Semantics of K)

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

- "Rewrite" rules over matching logic patterns:

$$\varphi \Rightarrow \varphi'$$

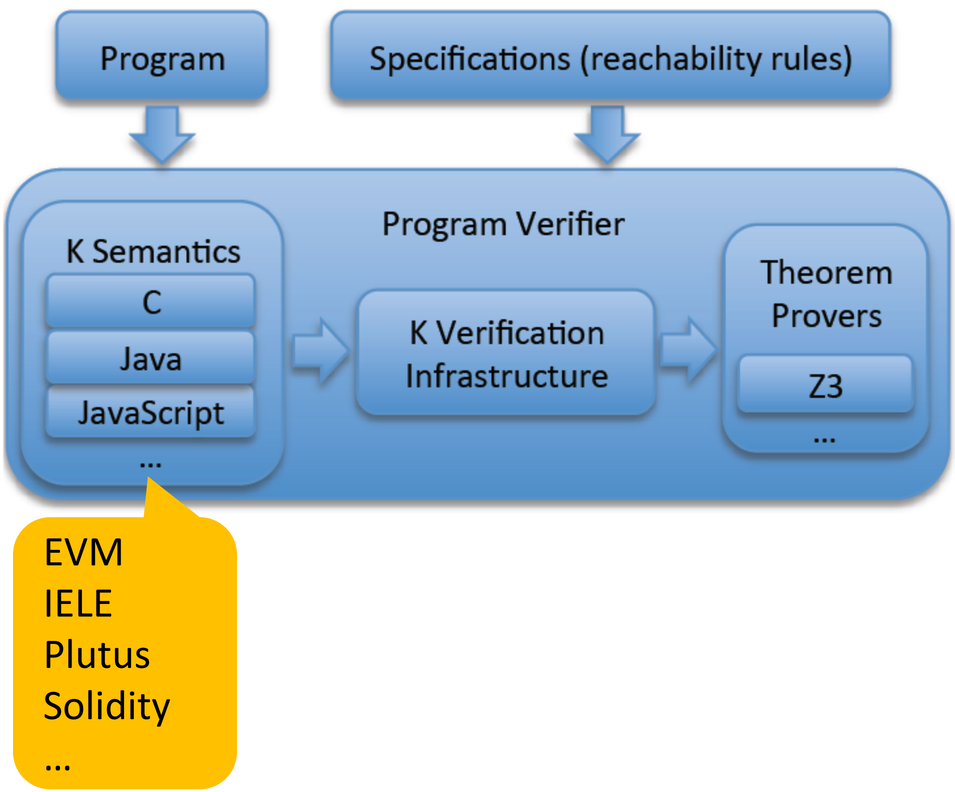
Can be expressed in matching logic:
$\varphi \rightarrow \Diamond(\varphi')$ $\Diamond$ is "weak eventually"

- 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 μ-logic theorems

K Deductive Program Verifier = = (Best Effort) Automation of MµL



- 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+...+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
```

Sum 1+2+…+n in IMP: Invariant

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

OK Performance
[OOPLSA'16]

Time (seconds) spent on applying semantic steps (symbolic execution)

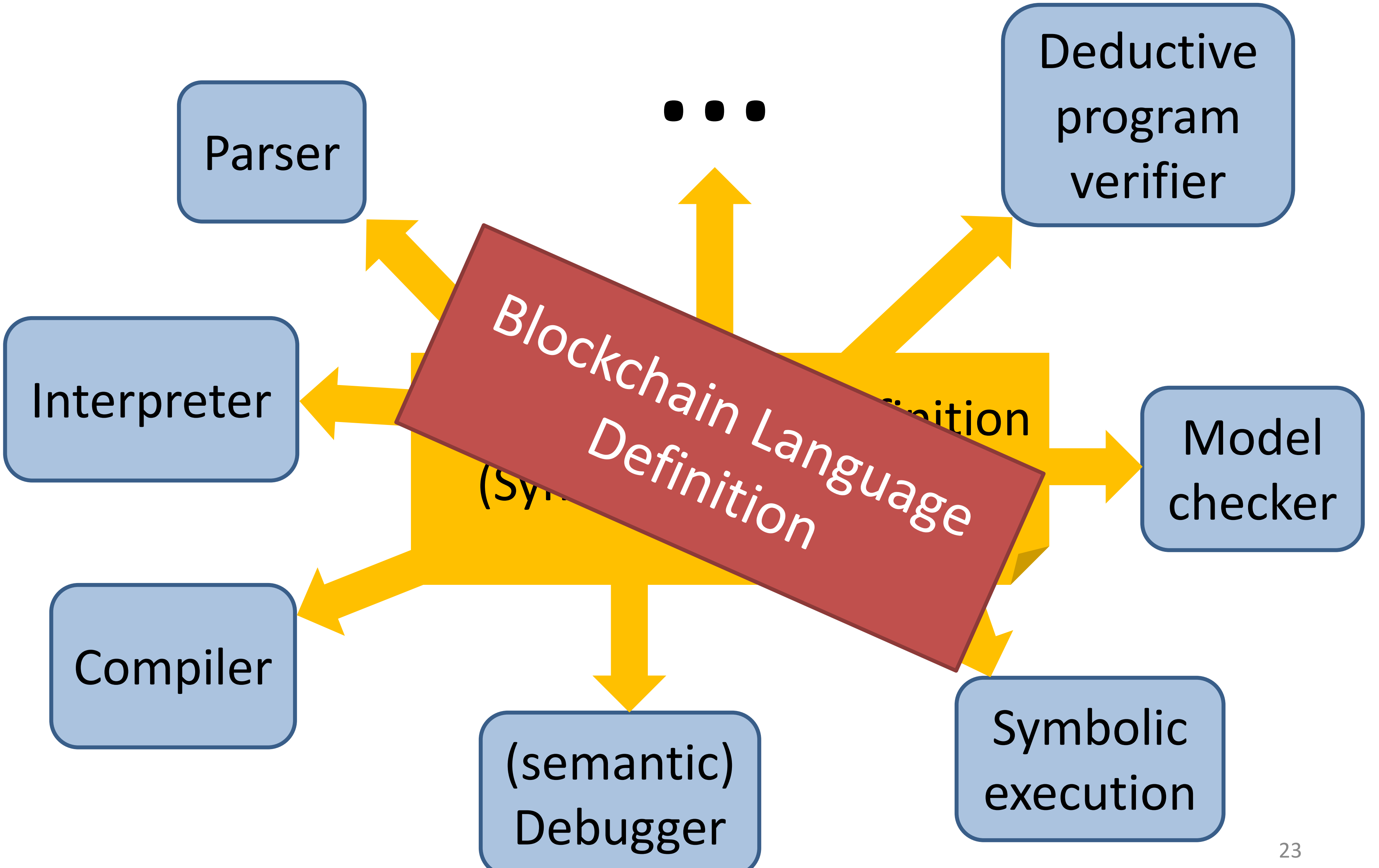
Time (seconds) spent on domain reasoning (matching logic + querying Z3)

Programs	KernelC				C				Java				JavaScript			
	Execution		Reasoning		Execution		Reasoning		Execution		Reasoning		Execution		Reasoning	
	Time	#Step	Time	#Query	Time	#Step	Time	#Query	Time	#Step	Time	#Query	Time	#Step	Time	#Query
BST find	0.6	192	1.2	95	10.4	1,028	3.6	246	1.9	322	2.8	244	4.5	1,736	1.8	93
BST insert	0.8	336	2.9	160	23.0	2,481	7.2	414	4.1	691	4.5	342	5.4	3,394	2.8	158
BST delete	1.4	582	5.6	420	55.1	4,540	16.6	938	9.8	1,274	15.1	1,125	15.6	5,052	5.6	373
AVL find	0.6	192	1.2	95	9.9	1,028	3.1	214	2.2	322	2.7	244	4.5	1,736	1.9	93
AVL insert	6.2	1,980	42.1	1,133	210.7	12,616	70.6	1,865	42.4	3,753	62.8	2,146	102.5	26,977	32.5	1,221
AVL delete	9.5	2,933	45.4	1,758	514.8	26,003	118.9	3,883	122.2	8,144	149.4	4,866	184.3	38,591	55.3	2,233
RBT find	0.6	192	1.1	95	11.5	1,064	3.0	214	2.1	322	2.9	244	4.9	1,736	1.9	93
RBT insert	7.6	2,331	48.1	1,392	722.0	30,924	181.8	4,394	39.9	4,240	75.7	2,547	84.9	28,082	29.6	1,381
RBT delete	10.6	3,891	33.7	2,033	1593.8	50,389	308.3	15,429	95.8	8,312	75.4	4,460	144.2	51,356	39.4	2,009
Treap find	0.6	200	1.4	118	11.2	1,064	3.2	214	2.0	322	2.9	244	4.6	1,736	1.9	116
Treap insert	1.4	753	4.5	247	52.4	4,954	15.3	724	12.7	1,469	10.4	563	13.7	7,738	5.2	243
Treap delete	2.0	831	9.4	509	73.9	5,512	16.5	656	12.0	1,694	16.4	1,021	24.8	8,333	8.4	460
List reverse	0.4	142	0.3	21	6.6	815	4.8	76	1.5	222	2.6	46	5.0	1,162	0.5	20
List append	0.4	171	0.5	45	7.4	909	7.4	128	1.8	239	5.5	106	4.5	1,392	0.8	46
Bubble sort	0.9	391	26.8	190	28.4	2,401	38.0	357	3.4	589	35.4	345	5.6	2,688	25.7	145
Insertion sort	1.1	468	24.5	300	26.6	2,555	35.3	451	4.1	731	27.0	371	8.3	3,119	36.5	213
Quick sort	1.1	604	31.6	269	31.0	3,601	48.2	518	7.1	958	40.0	413	15.0	5,046	33.1	252
Merge sort	1.7	970	55.0	478	81.6	6,589	89.0	1,070	14.1	1,566	72.9	737	22.8	7,021	43.2	480
Total	47.7	17,159	335.2	9,358	3470.5	158,473	970.6	31,791	379.3	35,170	604.5	20,064	654.9	196,895	326.3	9,629

- Properties very challenging to verify automatically. We only found one such prover for C, based on a separation logic extension of VCC
  - Which takes 260 sec to verify AVL insert (ours takes 280 sec; see above)

K for the Blockchain

K Framework Vision



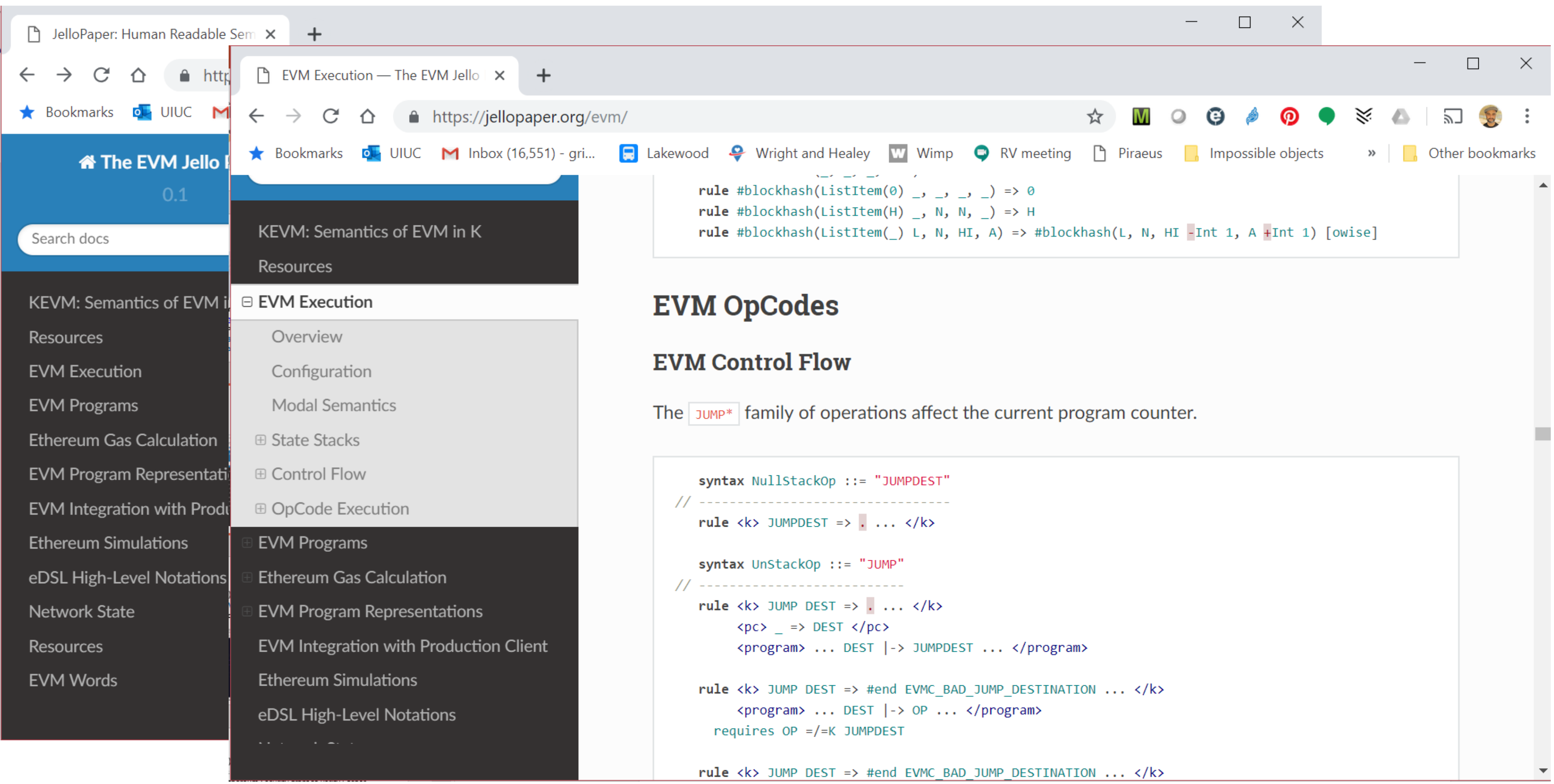
KEVM: Semantics of the Ethereum Virtual Machine (EVM) in K

[CSL'18]

Complete semantics of EVM in K
- https://github.com/kframework/evm-semantics
- Passes 60,000+ tests of C++ reference implementation
- *25% faster than ethereumjs, used by Truffle*
- *5x (only!) slower than ethereum-cpp*
- *Used as canonical EVM spec (replacing Yellow Paper)*

What Can We Do with KEVM?

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

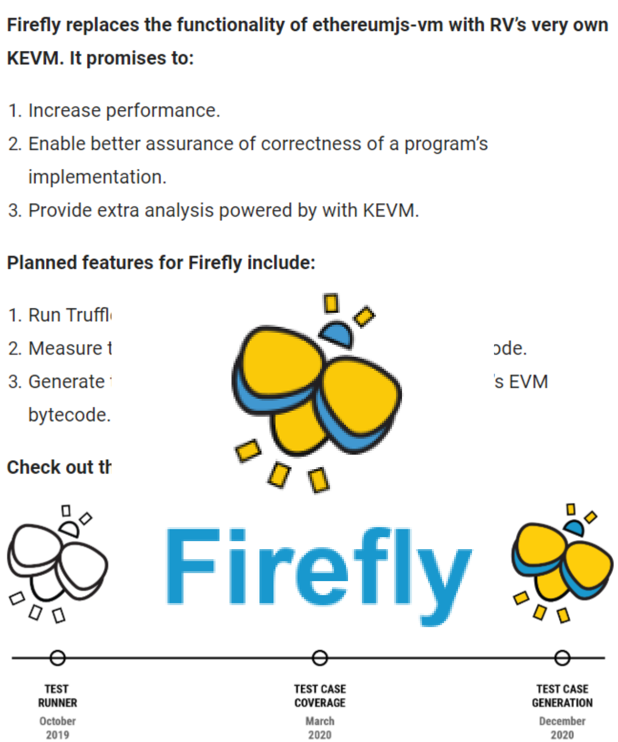


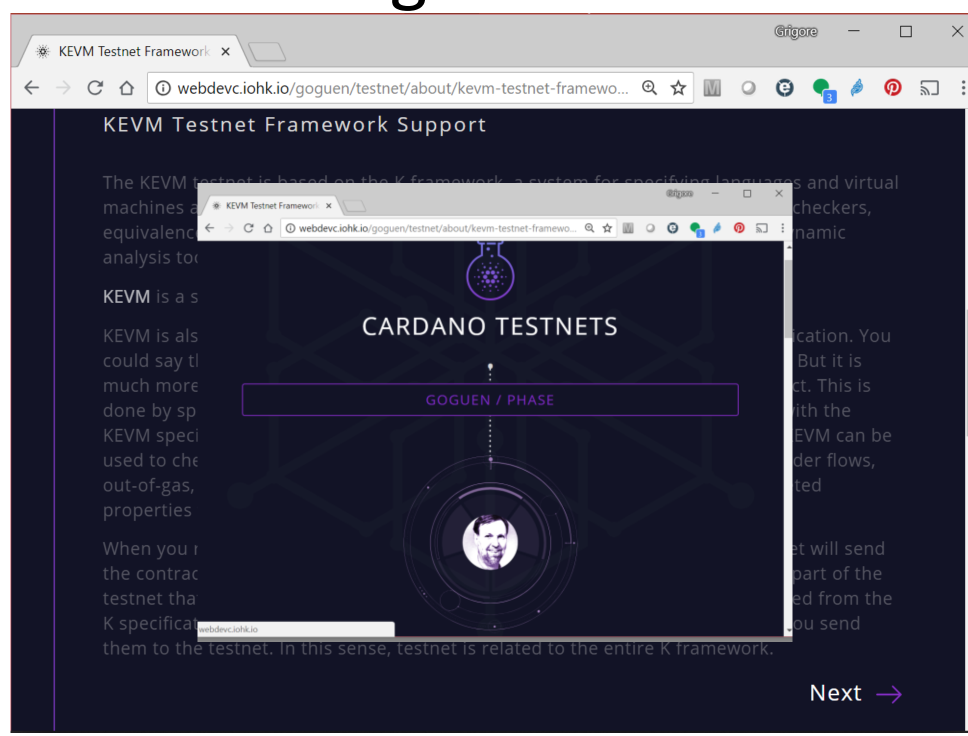
What Can We Do with KEVM?

2) *Generate and deploy correct-by-construction EVM client/simulator/emulator*

Firefly tool: KEVM to run, analyze and monitor tests

Cardano testnet: mantis executing KEVM

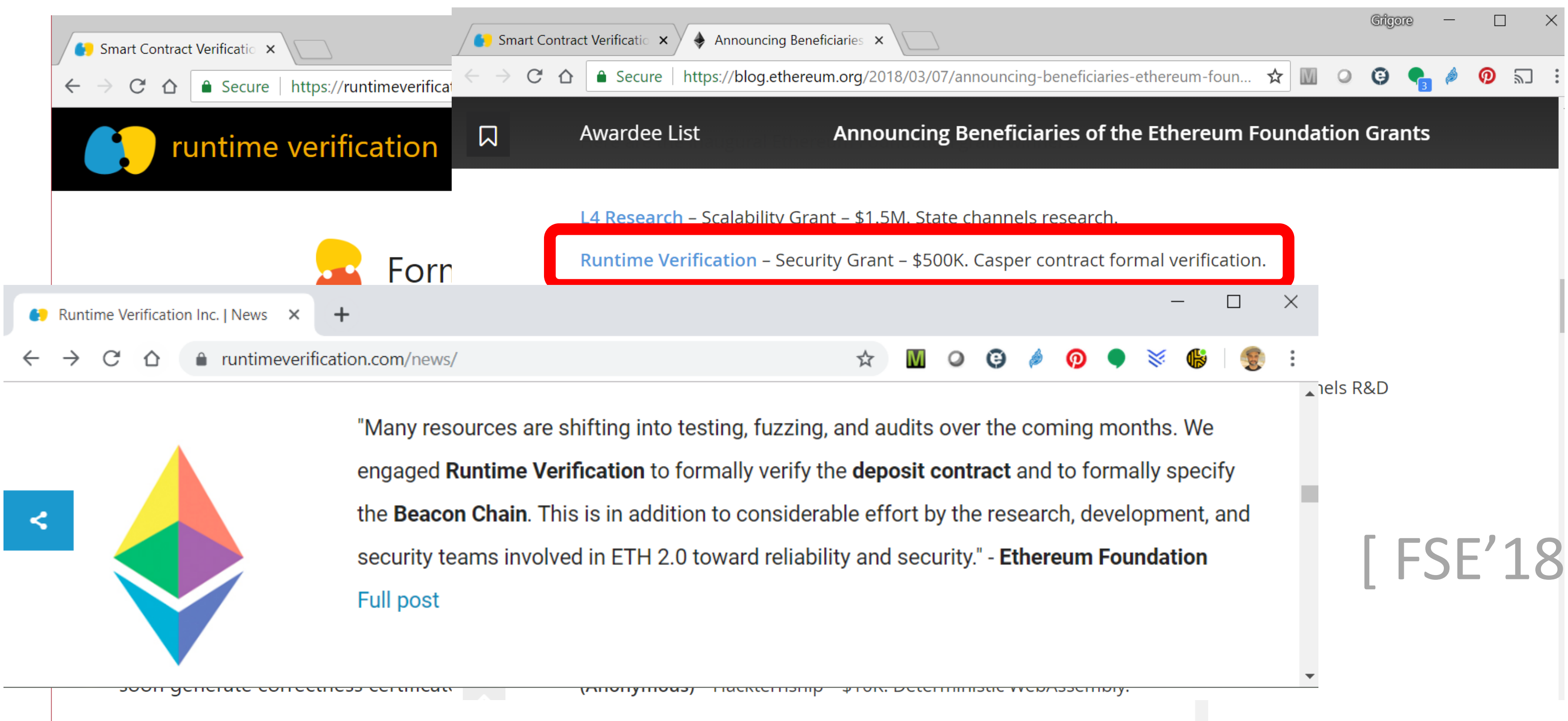




What Can We Do with KEVM?

3) *Formally verify Ethereum smart contracts*

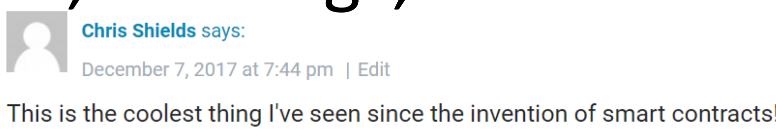
RV does that commercially. Won first Ethereum Security grant to verify Casper, then hired to formalize Beacon Chain (Serenity) and verify ETH1 -> ETH2 deposit contract



[FSE'18]

Formalizing ERC20, ERC777, … in K

- *K is very expressive for modeling*: languages, but also *token specifications and protocols; executable*
- 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, verified as EVM
- Others use or integrate our framework and specs:
  - DappHub (KLab), ETHWorks (Waffle), Consensys, Gnosis

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.

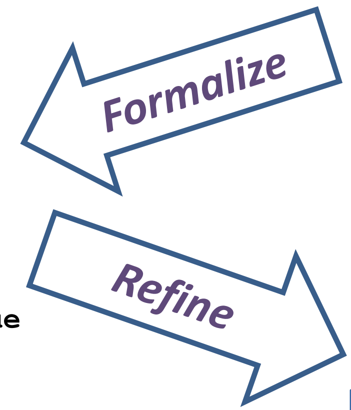
Note Transfers of 0 values MUST be treated as normal transfers and fire the `Transfer` event.

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

1 ERC20 Informal Business Logic

2 ERC20-K formal executable high-level spec

```
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
```



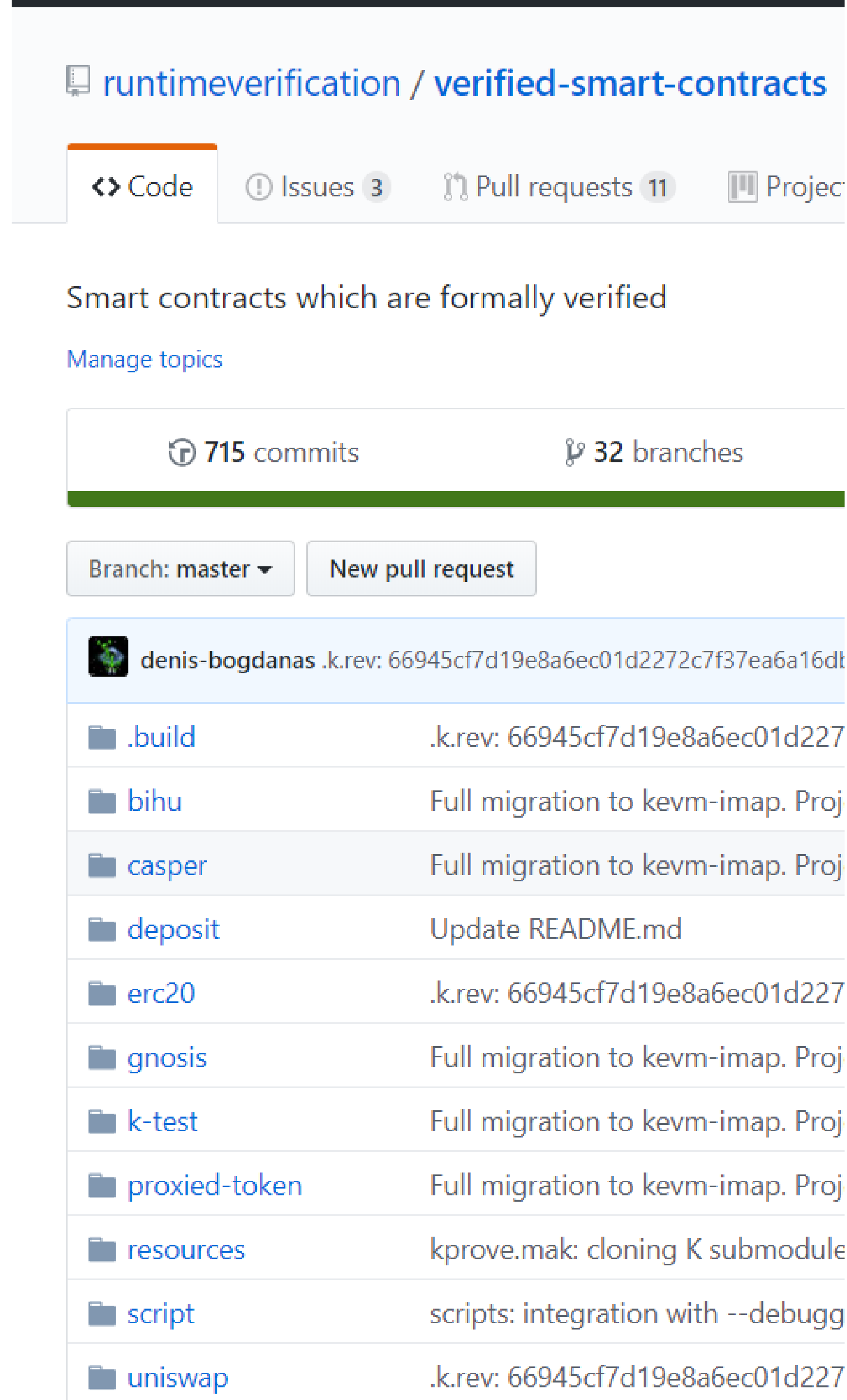
3 ERC20-EVM formal executable low-level spec that contains all EVM details

```
[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
……….
………
…….
```

Notable Contracts We've Verified

- ETH2.0 Deposit
- GnosisSafe
- Ethereum Casper FFG
- Uniswap
- DappSys DSToken ERC20
- Bihu KEY token



Designing New (and Better) Blockchain Languages Using K

EVM Not Human Readable (among other nuise

```
  PUSH(1, 0)  ; PUS
; PUSH(1, 10) ; PUS
; JUMPDEST
; PUSH(1, 0) ; PUSH
; ISZERO ; PUSH(1,
; PUSH(1, 32) ; MLO                 USH(1, 0)  ; MSTORE
; PUSH(1, 1)                        USH(1, 32) ; MSTORE
; PUSH(1, 10) ; JUM
; JUMPDEST
; PUSH(1, 0) ; MLOA
```

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
}
```

A New Virtual Machine (and Language) for the Blockchain

[FM'19]

- 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

Thanks to IOHK (iohk.io) for funding this project

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
- Vyper – in progress, collaboration with the Ethereum Foundation
- Plutus (functional) – collaboration with IOHK
- Flow (linear types, resources) – in progress, collaboration with DapperLabs (creators of CryptoKitties); plan is have *only* a K “implementation”

…

Modelling and Verification of Blockchain Protocols

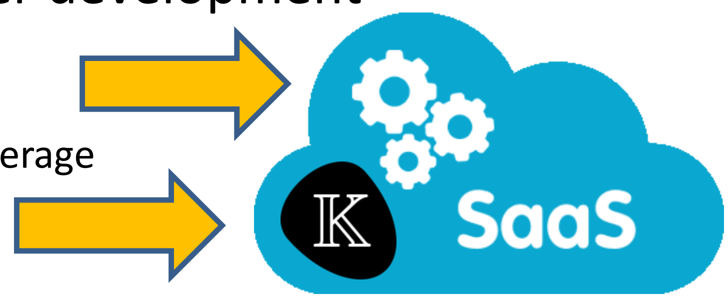
- Matching logic, rewriting and K can also be used to formally specify and verify consensus protocols, random number generators, etc.
- Done or ongoing:
  - Casper FFG (Ethereum Foundation)
  - RANDAO (Ethereum Foundation)
  - Algorand (Algorand)
  - Casper CBC (Coordination Technology)
  - Serenity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

K Blockchain Products and Tools in the Making. To be SaaS delivered

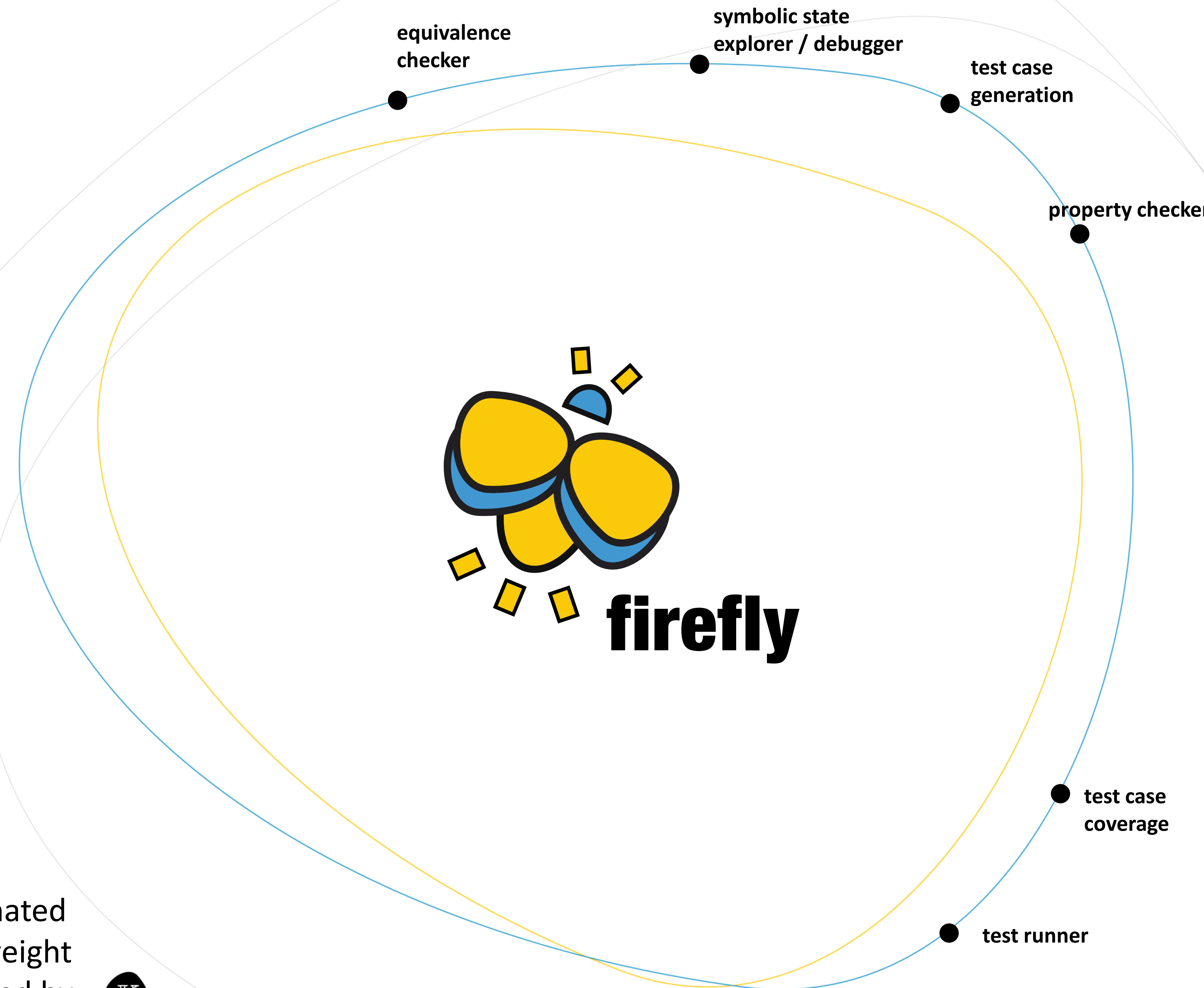
- Firefly – automated smart contract analysis
- KaaS – K formal verification as a service
- Proof objects – ultimate correctness certificates

Taking K to the Next Level

- Many people use K (40+ repositories and 50,000+ commits)
  - + Open source, used also for teaching PL at several universities
  - + Most comprehensive and rich in features language framework
  - – Hard to use and debug, poor error messages
  - – Slow (may take hours to formally verify non-trivial programs)
- Two major underlying engines under development
  1. *Concrete execution* engine (LLVM backend)
     - Many *parallel* calls in tools like test coverage
  2. *Symbolic execution* engine (Haskell backend)
     - Symbolic paths can be explored in *parallel*

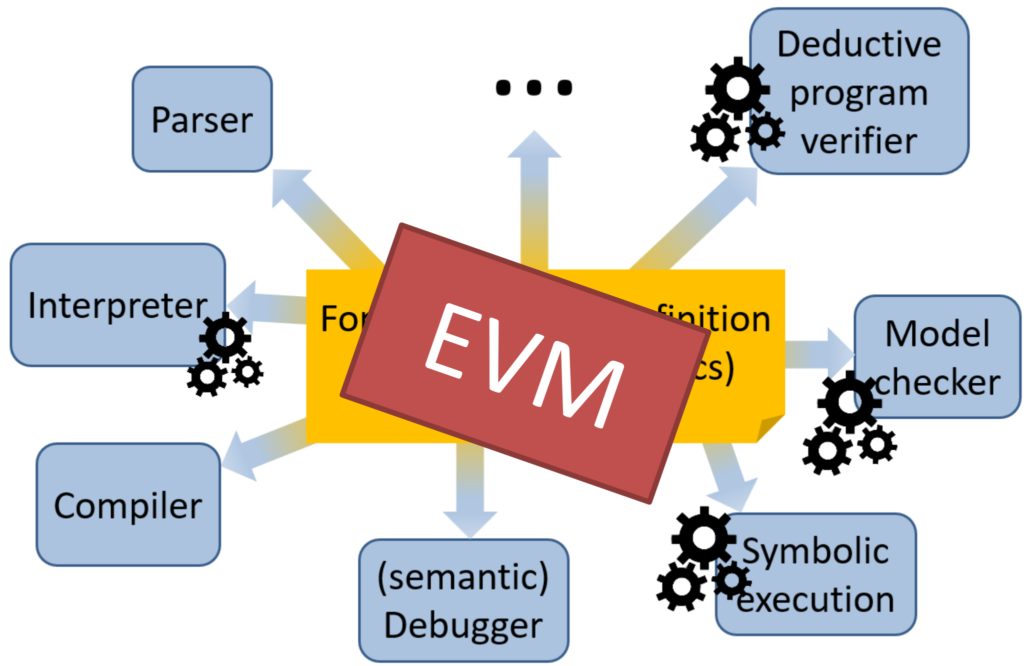


- Efficient implementations of these two engines will be offered as Software as a Service (SaaS) in the cloud
  - + Wait seconds, not minutes or hours!
  - + Good error messages, good debugger, good UX
  - + Proof objects, too (discussed shortly)



- Automated
- Lightweight
- Powered by K

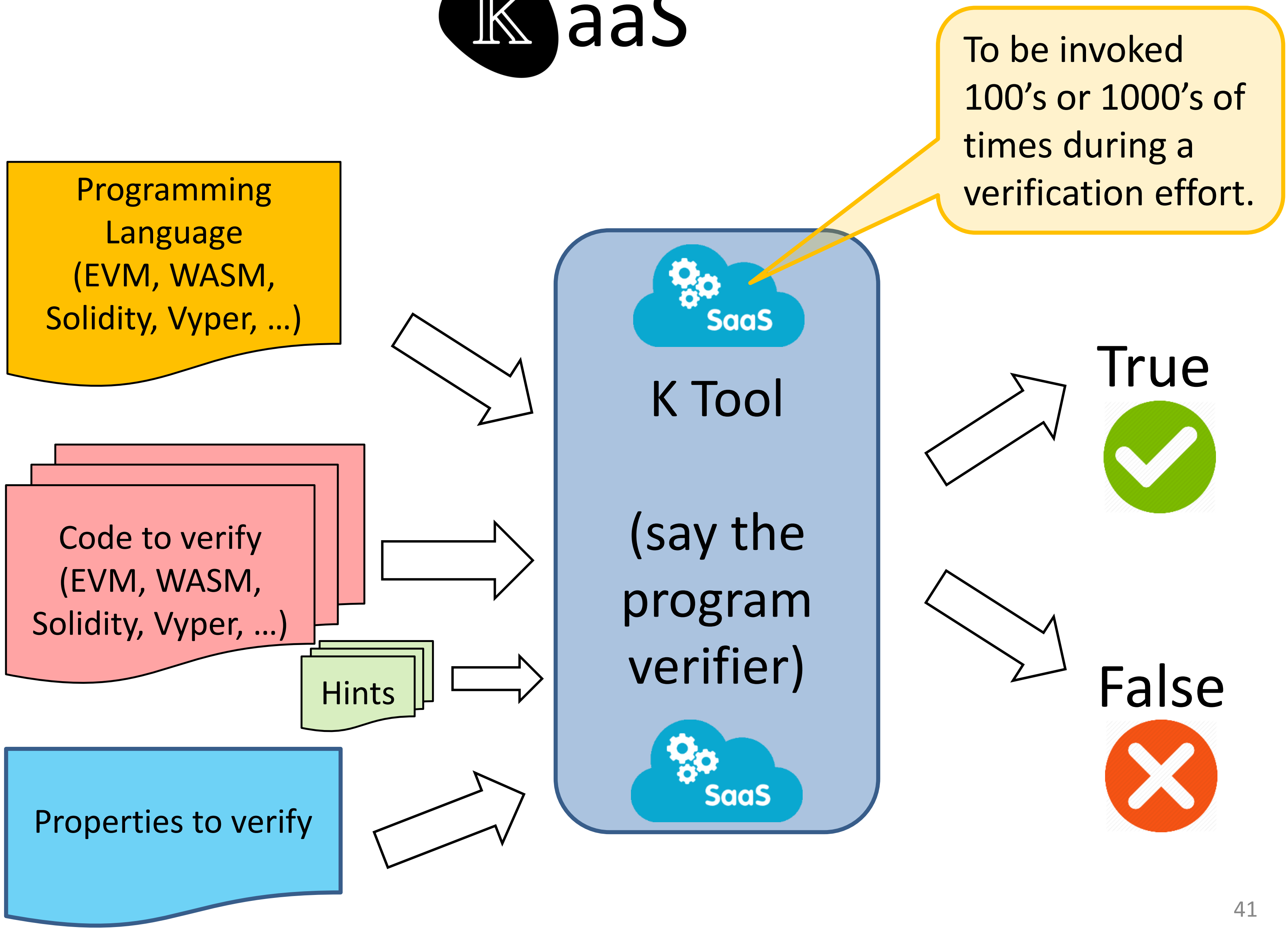
Firefly = K [EVM] + Automation



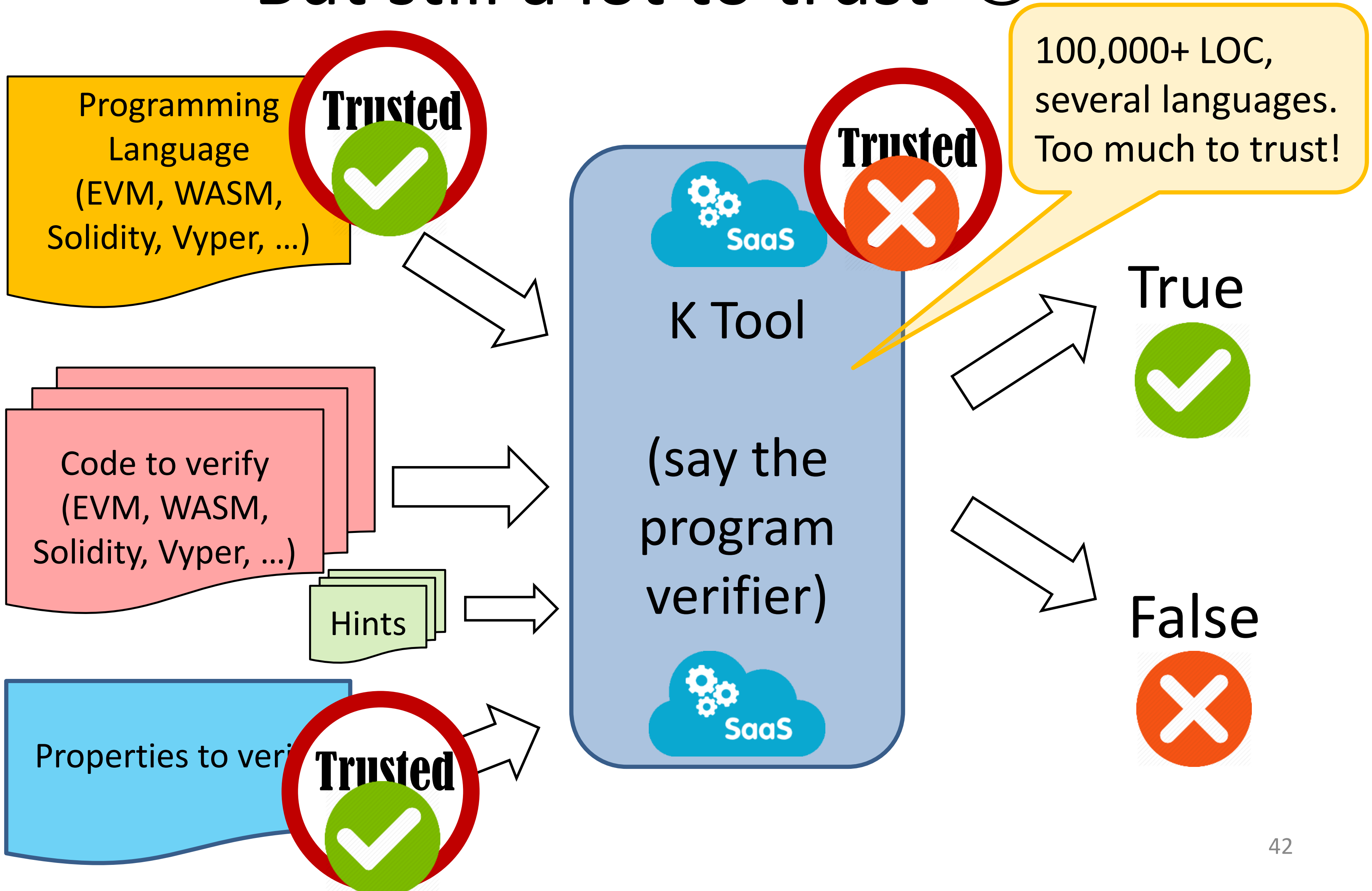
KaaS

formal verification
as a service

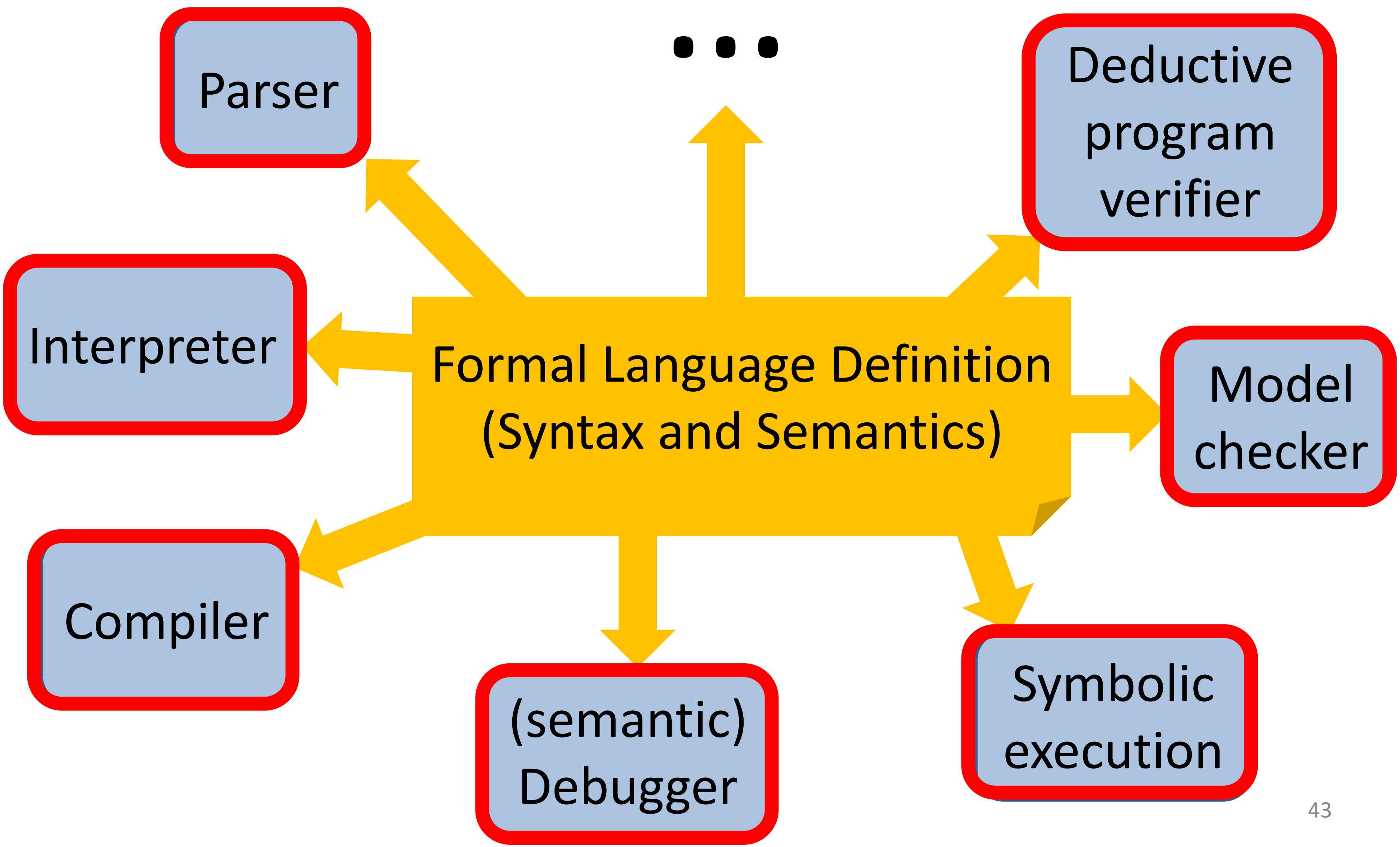
KaaS


To be invoked 100's or 1000's of times during a verification effort.
Programming Language (EVM, WASM, Solidity, Vyper, …)
Code to verify (EVM, WASM, Solidity, Vyper, …)
Hints
Properties to verify
SaaS
K Tool
(say the program verifier)
SaaS
True
False

Best Approach Ever! ☺
But still a lot to trust ☹



Proof Object Generation



Proof Object Generation

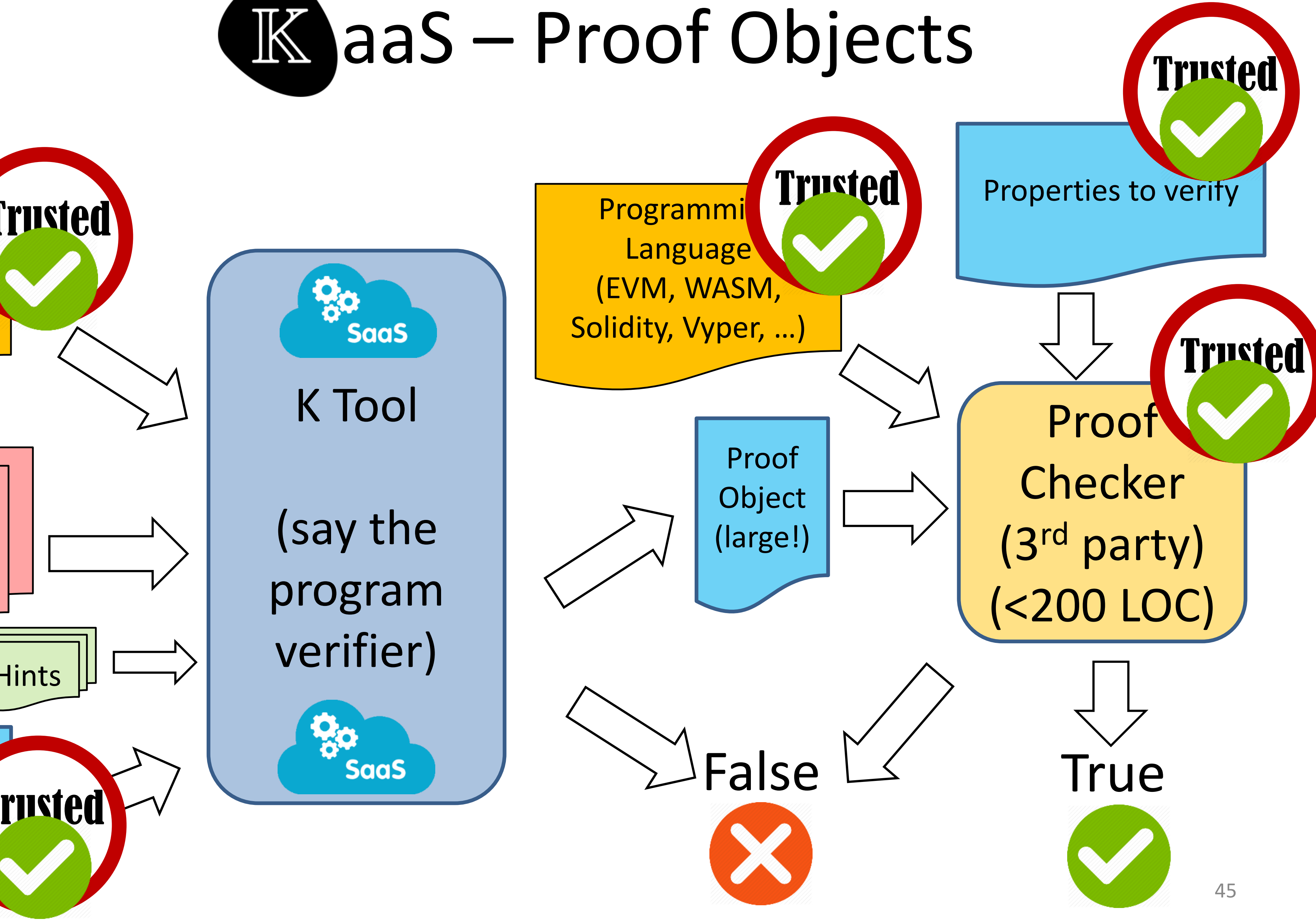
- Each of the K tools is a best-effort implementation of proof search in Matching μ-Logic:

$\mathcal{H}_\mu$, $\mathcal{H}$	(Proposition$_1$)	$\varphi_1 \to (\varphi_2 \to \varphi_1)$
	(Proposition$_2$)	$(\varphi_1 \to (\varphi_2 \to \varphi_3)) \to (\varphi_1 \to \varphi_2) \to (\varphi_1 \to \varphi_3)$
	(Proposition$_3$)	$(\neg\varphi_1 \to \neg\varphi_2) \to (\varphi_2 \to \varphi_1)$
	(Modus Ponens)	$\dfrac{\varphi_1 \quad \varphi_1 \to \varphi_2}{\varphi_2}$
	(Variable Substitution)	$\forall x.\varphi \to \varphi[y/x]$
	(∀)	$\forall x.(\varphi_1 \to \varphi_2) \to (\varphi_1 \to \forall x.\varphi_2)$ if $x \notin FV(\varphi_1)$
	(Universal Generalization)	$\dfrac{\varphi}{\forall x.\varphi}$
	(Propagation$_\bot$)	$C_\sigma[\bot] \to \bot$
	(Propagation$_\vee$)	$C_\sigma[\varphi_1 \vee \varphi_2] \to C_\sigma[\varphi_1] \vee C_\sigma[\varphi_2]$
	(Propagation$_\exists$)	$C_\sigma[\exists x.\varphi] \to \exists x.C_\sigma[\varphi]$ if $x \notin FV(C_\sigma[\exists x.\varphi])$
	(Framing)	$\dfrac{\varphi_1 \to \varphi_2}{C_\sigma[\varphi_1] \to C_\sigma[\varphi_2]}$
	(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.
	(Set Variable Substitution)	$\dfrac{\varphi}{\varphi[\psi/X]}$
	(Pre-Fixpoint)	$\varphi[\mu X.\,\varphi/X] \to \mu X.\varphi$
	(Knaster-Tarski)	$\dfrac{\varphi[\psi/X] \to \psi}{\mu X.\,\varphi \to \psi}$

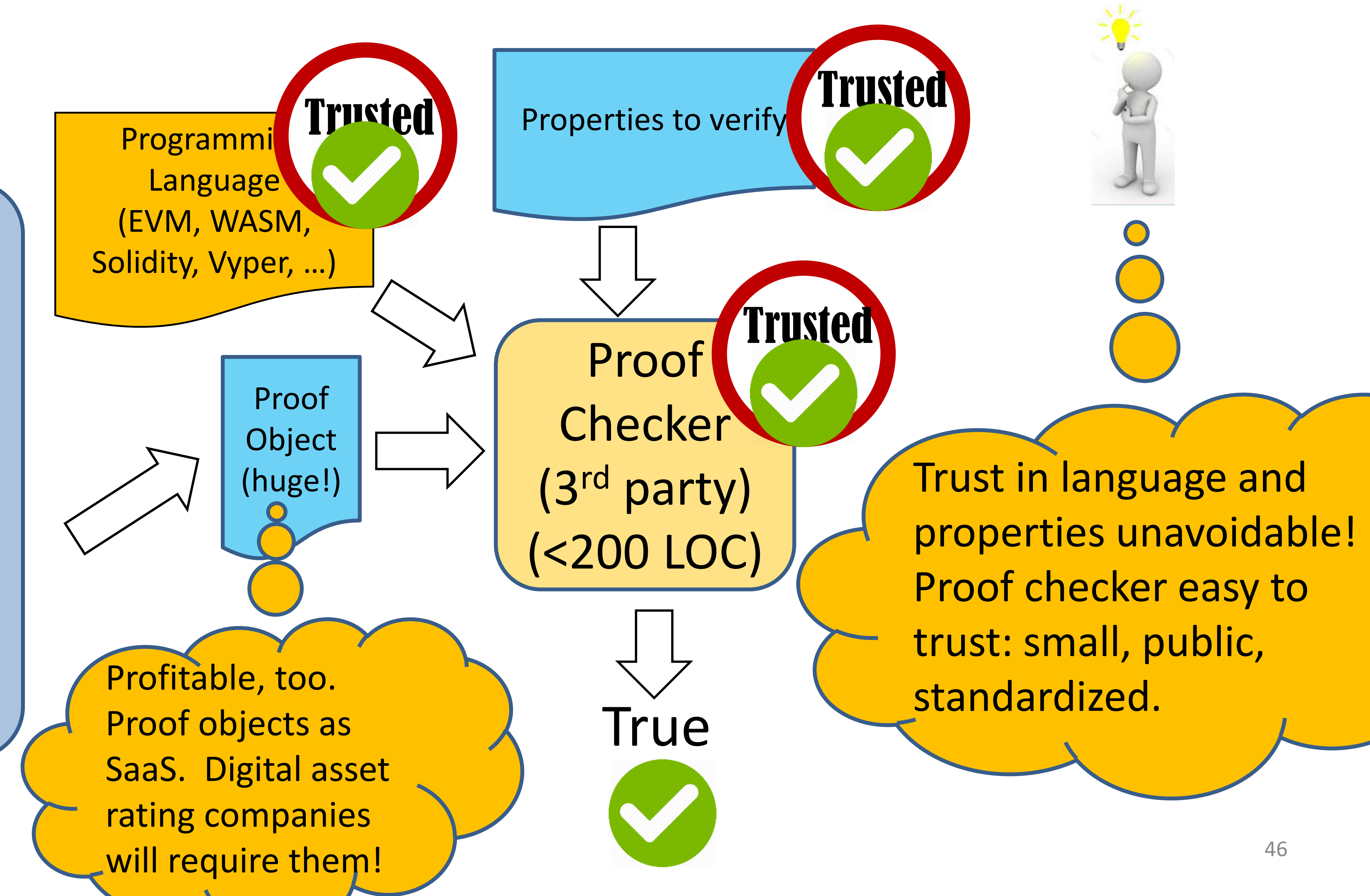
16 proof rules only!
Simple proof checker (<200 LOC)!
In contrast, Coq has about 45 proof rules, and its proof checker has 8000+ lines of OCAML

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

KaaS – Proof Objects



Assured Trust. Like Never Before!

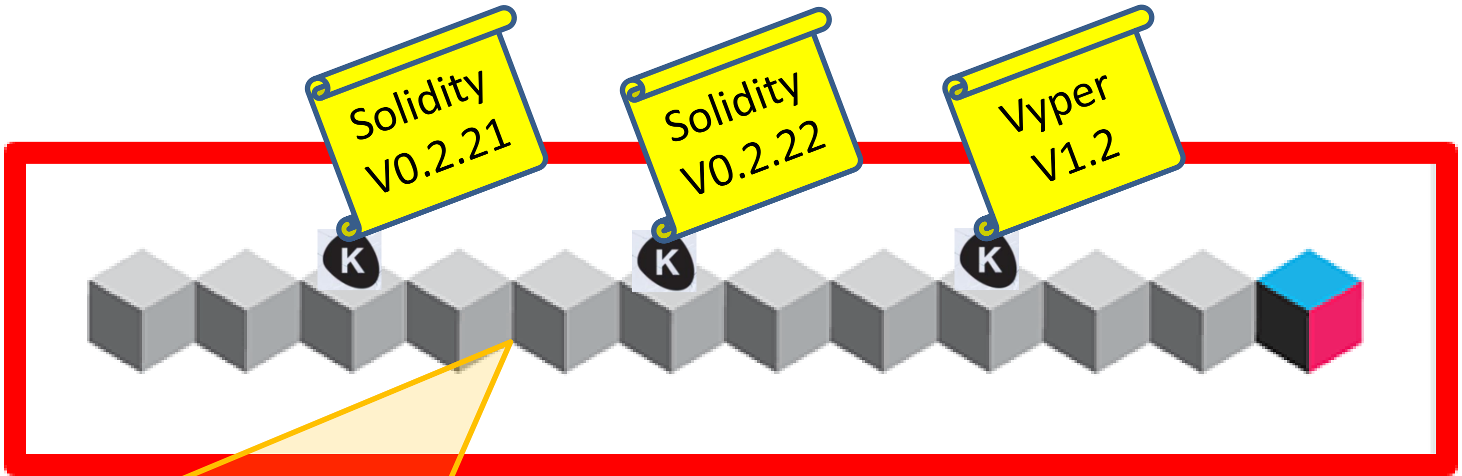


K - Powered

Blockchain

K as 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 can be used as execution engine / VM.

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 -

```
pragma solidity ^0.5.0;

import "./IERC20.sol";
import "../../math/SafeMath.sol";

contract ERC20 is IERC20 {
    using SafeMath for uint256;

    mapping (address => uint256) private _balances;

    function transfer(address to, uint256 value) public returns (bool) {
        _transfer(msg.sender, to, value);
        return true;
    }

    function _transfer(address from, address to, uint256 value) internal {
        require(to != address(0), "ERC20: transfer to the zero address");

        _balances[from] = _balances[from].sub(value);
        _balances[to] = _balances[to].add(value);
        emit Transfer(from, to, value);
    }

}
```

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
...
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF AND DUP4 PUSH20 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF 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

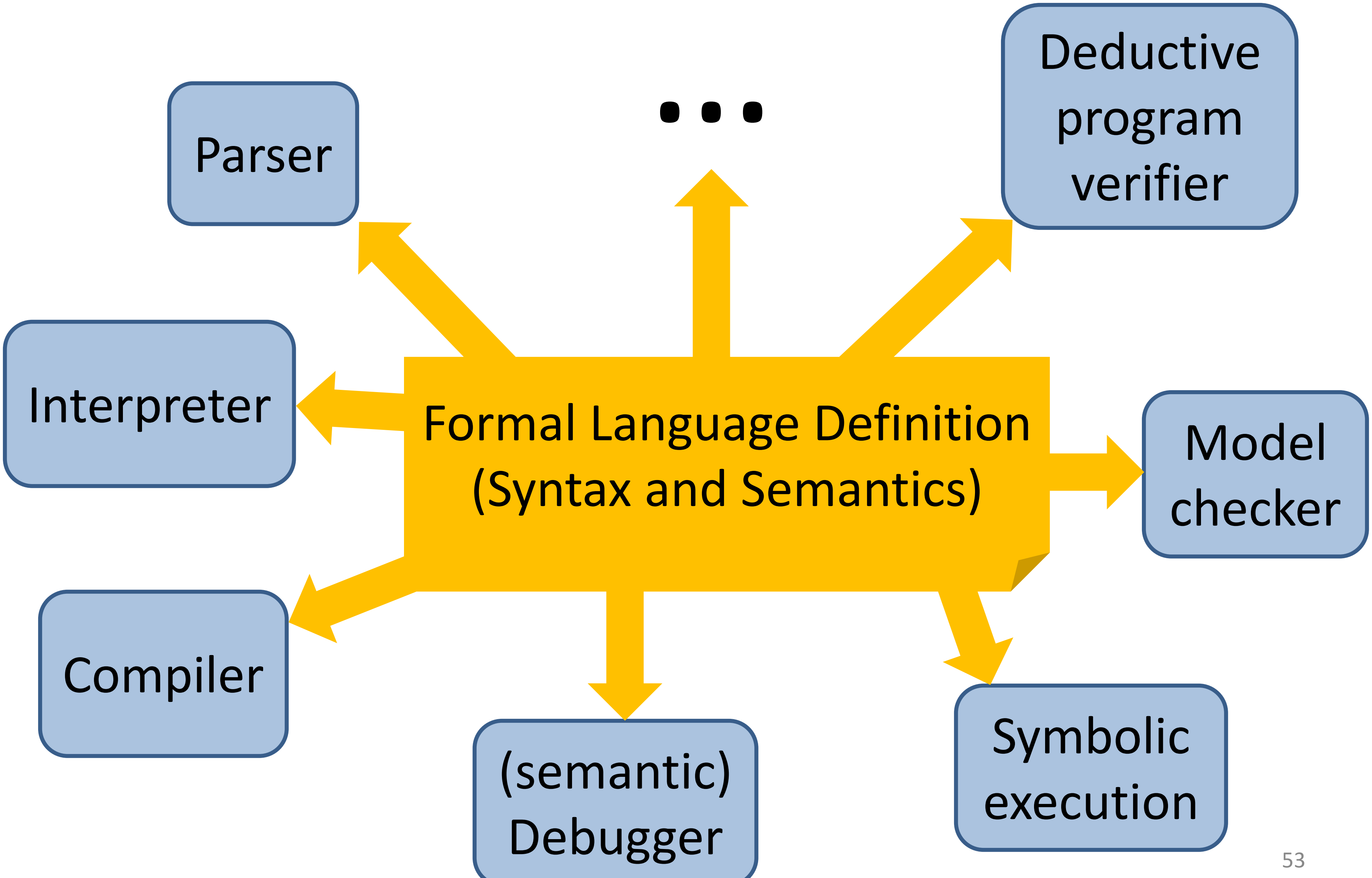
- Snippet, Sugared -

```
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
- *Use K as programming language for smart contracts!*

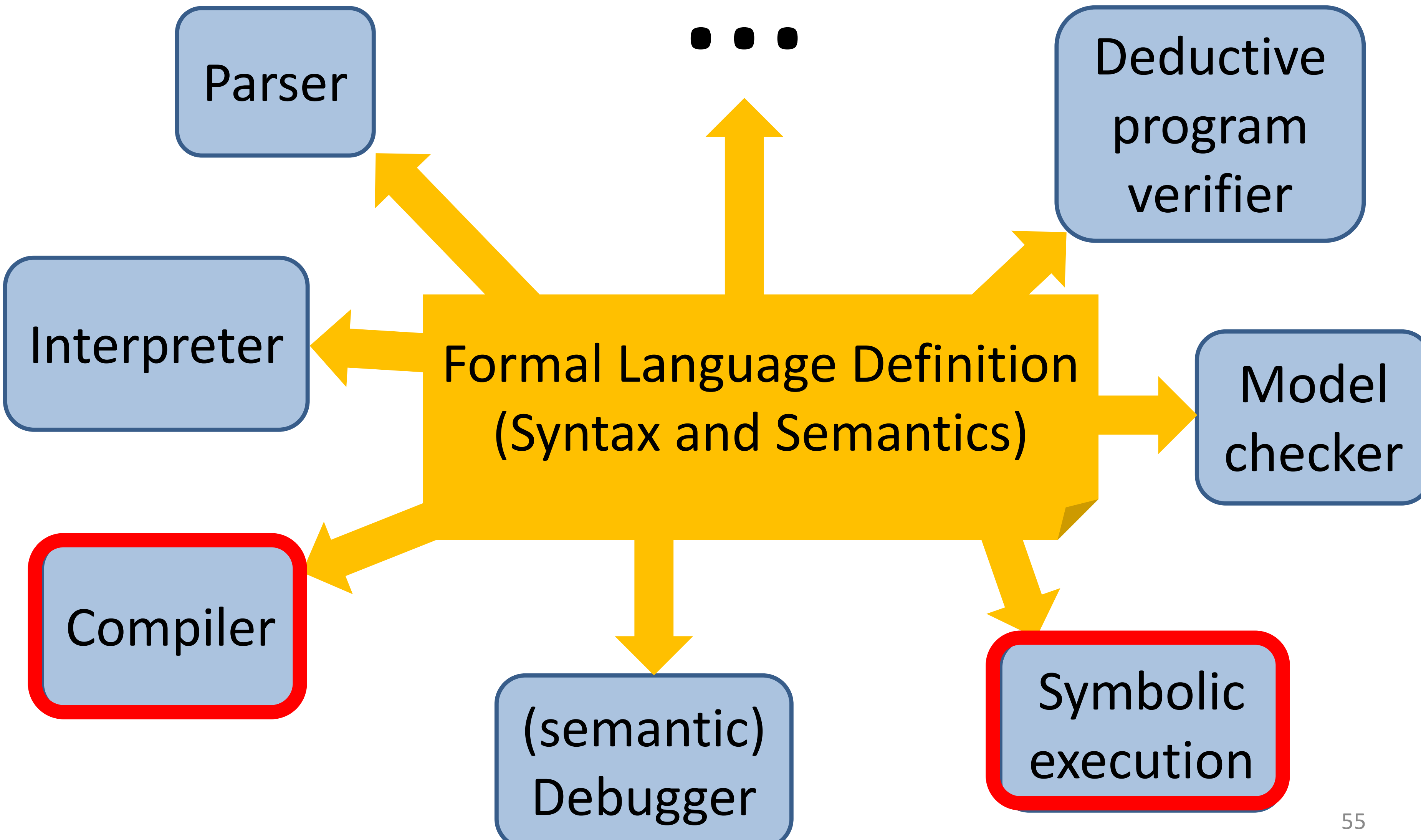
(needed: gas model for K)

Conclusion: It Can Be Done!

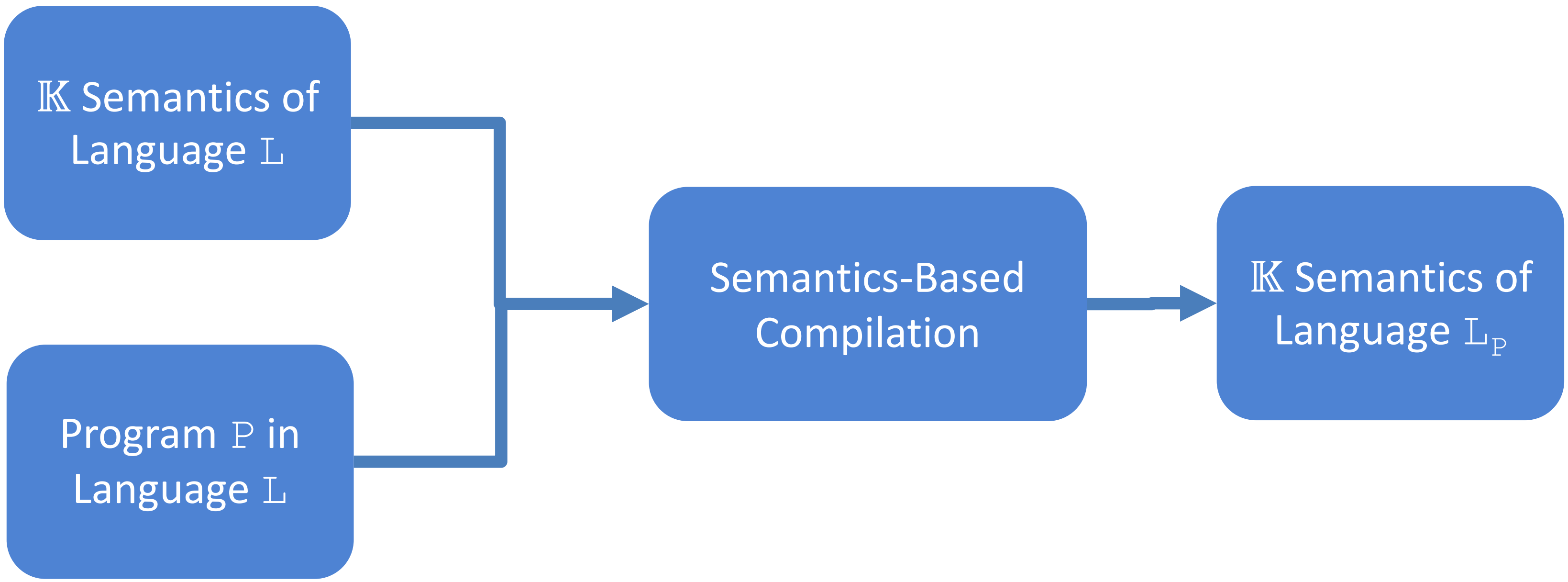


Extra Slides

Semantics-Based Compilation



Semantics-Based Compilation (SBC)



Goals

- Execution of P in L equivalent to executing L_P in a start configuration
- L_P should be "as simple as possible", only capturing exactly the dynamics of L necessary to execute program P

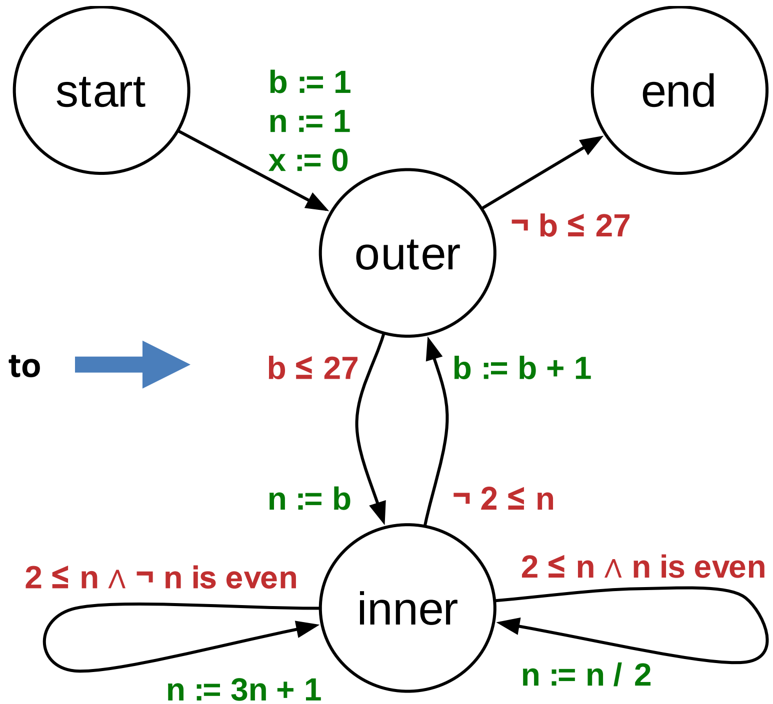
Semantics-Based Compilation (SBC) Experiments with Early Prototype

```
// 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
```

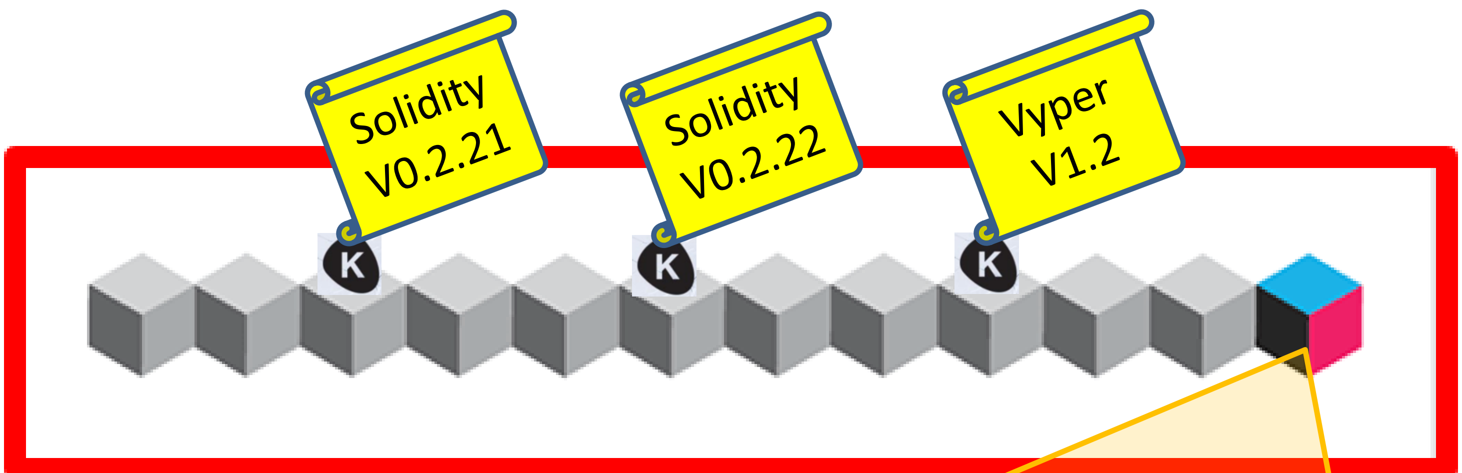
compiles to



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

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