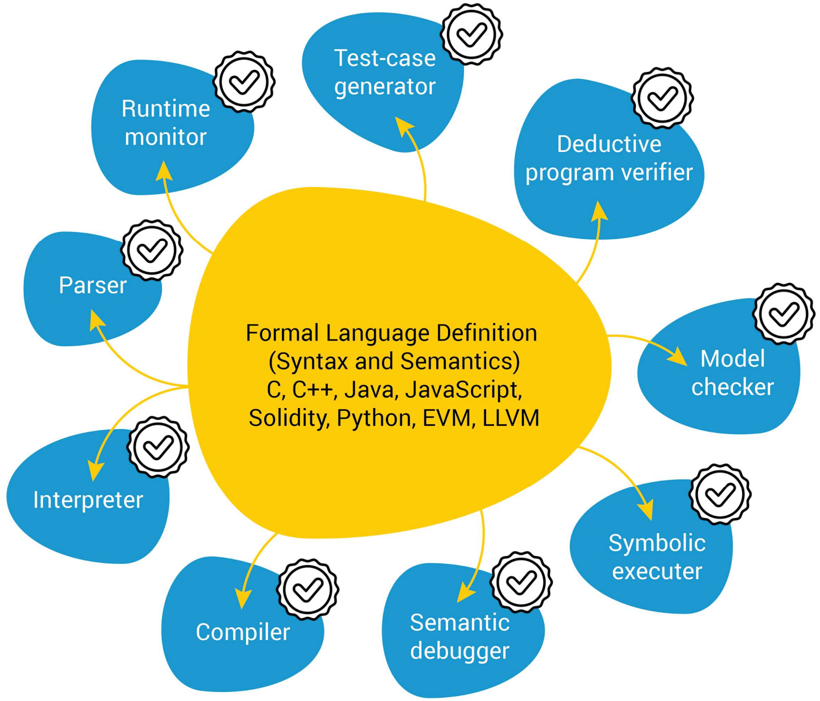
Towards a Trustworthy Language Framework via Proof Generation

Xiaohong Chen, Zhengyao Lin, Minh-Thai Trinh, Grigore Rosu
University of Illinois at Urbana-Champaign
{xc3,zl38,trinhmt,grosu}@illinois.edu

2021 July

Overview

- We turn **<u>program execution</u>** into **<u>mathematical proofs</u>**.
  - Rigorous, complete, machine-checkable proofs
  - A very small, 245-LOC trust base
- Motivation: A language framework
  - K framework (https://kframework.org)
- Correctness by proofs, case-by-case:
  - Give one execution trace
  - Generate a proof of that trace
- A prototype implementation
  - OK proof generation time (minutes)
  - fast proof checking time (seconds)
  - very large proof objects (millions LOC)

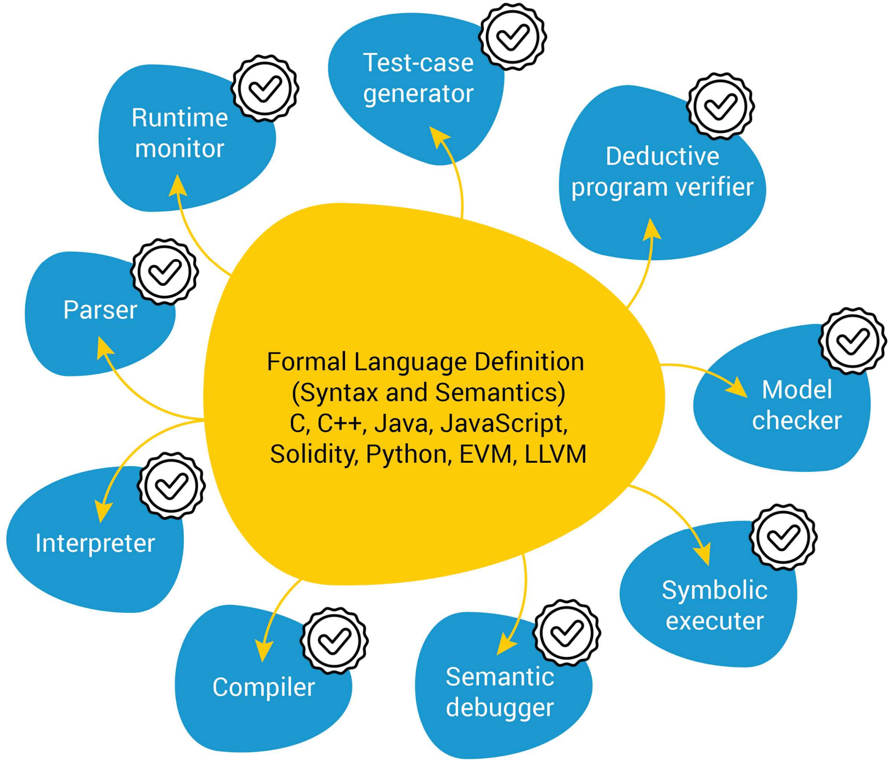


Outline

- K framework (https://kframework.org)
- Logical foundation of K
  - Matching logic (http://matching-logic.org)
- Turn a trace: $t_0, t_1, t_2, \ldots, t_n$ (of language L) into a proof $\boxed{\Gamma^L \vdash t_0 \Rightarrow t_n}$
  - Formalize/encode matching logic and "$\vdash$"
  - Translate K formal semantics into Γ^L
  - Generate the proofs
- Implementation & Experiment

K Overview

- https://kframework.org
- K = a meta-language to define PLs
  - C, Java, JavaScript, Ethereum VM, Python, Rust, x86-64, etc
- **<u>Language-independence</u>**
  - Proof generation for all languages!



K vision

An Example of K

```
module IMP-SYNTAX
  imports DOMAINS-SYNTAX
  syntax Exp ::=
      Int
    | Id
    | Exp "+" Exp      [left, strict]
    | Exp "-" Exp      [left, strict]
    | "(" Exp ")"      [bracket]
  syntax Stmt ::=
      Id "=" Exp ";"   [strict(2)]
    | "if" "(" Exp ")"
        Stmt Stmt      [strict(1)]
    | "while" "(" Exp ")" Stmt
    | "{" Stmt "}"     [bracket]
    | "{" "}"
    > Stmt Stmt        [left, strict(1)]
  syntax Pgm ::= "int" Ids ";" Stmt
  syntax Ids ::= List{Id,","}
endmodule
```

PL syntax

```
module IMP imports IMP-SYNTAX
  imports DOMAINS
  syntax KResult ::= Int
  configuration
   <T> <k> $PGM:Pgm </k>
       <state> .Map </state> </T>
  rule <k> X:Id => I ...</k>
       <state>... X |-> I ...</state>
  rule I1 + I2 => I1 +Int I2
  rule I1 - I2 => I1 -Int I2
  rule <k> X = I:Int => I ...</k>
       <state>... X |-> (_ => I) ...</state>
  rule {} S:Stmt => S
  rule if(I) S _ => S requires I =/=Int 0
  rule if(0) _ S => S
  rule while(B) S => if(B) {S while(B) S} {}
  rule <k> int (X, Xs => Xs) ; S </k>
       <state>... (. => X |-> 0) </state>
  rule int .Ids ; S => S
endmodule
```

PL configurations (program code + states)

PL semantics (rewrite rules)

Fig. 2: The complete 𝕂 formal definition of an imperative language IMP.

Use K to Execute Programs

- Only one rewrite rule:
  $\langle m, n \rangle \Rightarrow \langle m-1, n+m \rangle$ if $m > 0$

$$\langle 100, 0\rangle, \langle 99, 100\rangle, \langle 98, 199\rangle, \ldots, \langle 1, 5049\rangle, \langle 0, 5050\rangle$$

```
module TWO-COUNTERS
  imports INT
  syntax State ::= "<" Int "," Int ">"
  configuration <T> $PGM:State </T>
  rule <M, N> => <M -Int 1, N +Int M>
       requires M >Int 0
endmodule
```

Fig. 3: Running example `TWO-COUNTERS`.

- To make K generate the above trace
  - Put ⟨100,0⟩ in a source file, say `100.two-counters`
  - Compile the K semantics into a matching logic theory
    `$ kompile two-counters.k`
  - Call K execution tool
    `$ krun 100.two-counters --depth N`

run N steps, so we get the execution trace (by letting N=0,1,2,⋯)

Logical Foundation of K

- Matching logic (http://matching-logic.org)
  - K semantics = matching logic theory. E.g., $\Gamma^{two-counters}$
  - K tools = matching logic proofs. E.g., $\Gamma^{two-counters} \vdash \langle 100,0 \rangle \Rightarrow \langle 0,5050 \rangle$
- Simple syntax. Simple proof system (next slide)

$$\text{patterns} \quad \varphi ::= \underbrace{\boxed{x \mid X \mid \sigma \mid \varphi_1\ \varphi_2}}_{\text{variables, symbols, and application}} \mid \underbrace{\boxed{\bot \mid \varphi_1 \to \varphi_2}}_{\text{propositional logic}} \mid \underbrace{\boxed{\exists x.\, \varphi}}_{\text{quantification}} \mid \underbrace{\boxed{\mu X.\, \varphi}}_{\text{fixpoints \& induction}}$$

- Expressive. Complex concepts defined by **axioms/theories**
  - theories of equality $\Gamma^{equality}$, of sorts/types Γ^{sorts}, of rewriting $\Gamma^{rewriting}$
  - theory of a PL $\Gamma^{L} \supseteq \Gamma^{equality} \cup \Gamma^{sorts} \cup \Gamma^{rewriting} \cup \cdots$

Matching Logic Proof System

- Defines provability $\Gamma \vdash \varphi$
- Hilbert-style proof system
- 15 simple proof rules
  - Easy to implement
  - Small trust base

- Formalize matching logic
  - Its syntax
  - Its proof rules

FOL Rules	(Propositional 1)	$\varphi \to (\psi \to \varphi)$
	(Propositional 2)	$(\varphi \to (\psi \to \theta)) \to ((\varphi \to \psi) \to (\varphi \to \theta))$
	(Propositional 3)	$((\varphi \to \bot) \to \bot) \to \varphi$
	(Modus Ponens)	$\dfrac{\varphi \quad \varphi \to \psi}{\psi}$
	($\exists$-Quantifier)	$\varphi[y/x] \to \exists x .\, \varphi$
	($\exists$-Generalization)	$\dfrac{\varphi \to \psi}{(\exists x .\, \varphi) \to \psi}\ x \notin FV(\psi)$
Frame Rules	(Propagation$_\bot$)	$C[\bot] \to \bot$
	(Propagation$_\vee$)	$C[\varphi \vee \psi] \to C[\varphi] \vee C[\psi]$
	(Propagation$_\exists$)	$C[\exists x .\, \varphi] \to \exists x .\, C[\varphi]$ with $x \notin FV(C)$
	(Framing)	$\dfrac{\varphi \to \psi}{C[\varphi] \to C[\psi]}$
Fixpoint Rules	(Substitution)	$\dfrac{\varphi}{\varphi[\psi/X]}$
	(Prefixpoint)	$\varphi[(\mu X .\, \varphi)/X] \to \mu X .\, \varphi$
	(Knaster-Tarski)	$\dfrac{\varphi[\psi/X] \to \psi}{(\mu X .\, \varphi) \to \psi}$
Technical Rules	(Existence)	$\exists x .\, x$
	(Singleton)	$\neg(C_1[x \wedge \varphi] \wedge C_2[x \wedge \neg\varphi])$

Fig. 5: Matching logic proof system (where C, C_1, C_2 are application contexts).

Formalization of Matching Logic

- We use Metamath
  - http://metamath.org
  - A tiny language to encode formal systems and proofs
  - Very fast and simple proof verifying
- Matching logic defined in **245 lines** of Metamath
  - Very small trust base

logic syntax

```
$c \imp ( ) #Pattern |- $.

$v ph1 ph2 ph3 $.
ph1-is-pattern $f #Pattern ph1 $.
ph2-is-pattern $f #Pattern ph2 $.
ph3-is-pattern $f #Pattern ph3 $.
imp-is-pattern
  $a #Pattern ( \imp ph1 ph2 ) $.
```

```
axiom-1
  $a |- ( \imp ph1 ( \imp ph2 ph1 ) ) $.

axiom-2
  $a |- ( \imp ( \imp ph1 ( \imp ph2 ph3 ) )
        ( \imp ( \imp ph1 ph2 )
              ( \imp ph1 ph3 ) ) ) $.

${
  rule-mp.0 $e |- ( \imp ph1 ph2 ) $.
  rule-mp.1 $e |- ph1 $.
  rule-mp   $a |- ph2 $.
$}
```

proof rules

(meta-)theorems

```
imp-refl $p |- ( \imp ph1 ph1 )
$=
  ph1-is-pattern ph1-is-pattern
  ph1-is-pattern imp-is-pattern
  imp-is-pattern ph1-is-pattern
  ph1-is-pattern imp-is-pattern
  ph1-is-pattern ph1-is-pattern
  ph1-is-pattern imp-is-pattern
  ph1-is-pattern imp-is-pattern
  imp-is-pattern ph1-is-pattern
  ph1-is-pattern ph1-is-pattern
  imp-is-pattern imp-is-pattern
  ph1-is-pattern ph1-is-pattern
  imp-is-pattern imp-is-pattern
  ph1-is-pattern ph1-is-pattern
  ph1-is-pattern imp-is-pattern
  ph1-is-pattern axiom-2
  ph1-is-pattern ph1-is-pattern
  ph1-is-pattern imp-is-pattern
  axiom-1 rule-mp ph1-is-pattern
  ph1-is-pattern axiom-1 rule-mp
$.
```

The Metamath encoding of a formal proof

Fig. 6: An extract of the Metamath formalization of matching logic.

$\underbrace{\hspace{6em}}_{\text{trust base}}$ $\underbrace{\hspace{6em}}_{\text{not in trust base}}$

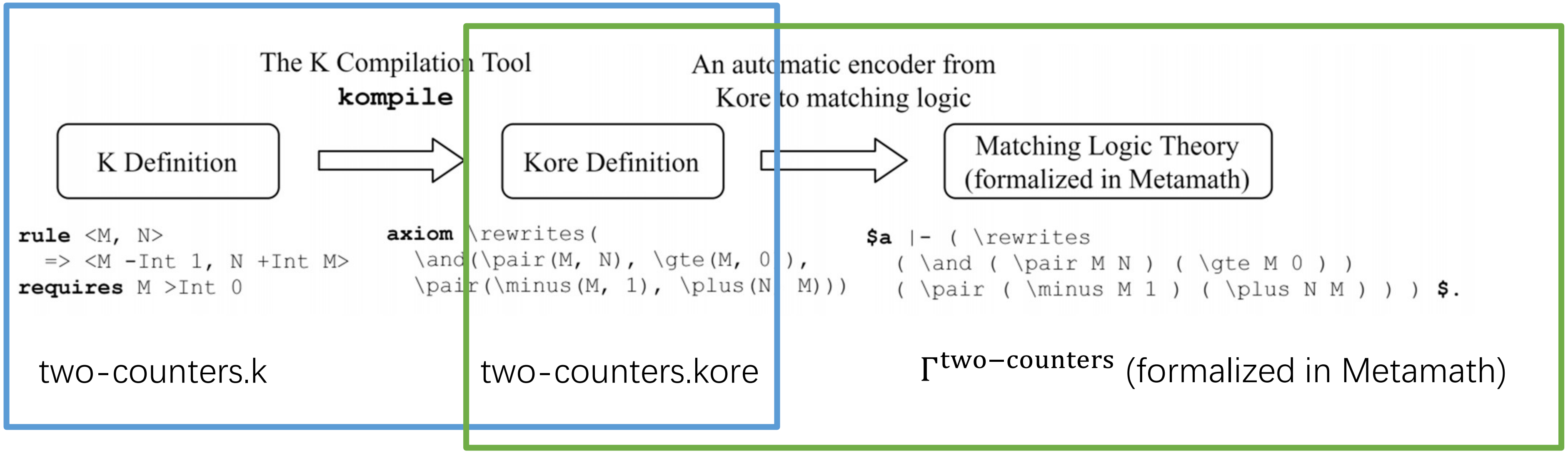
Formalization of Matching Logic

- **Within the 245-line trust base:**
  - logic syntax and proof system
  - metalevel operations (fresh variables, substitution, etc.)
  - support for notations (e.g., $\neg\varphi \equiv \varphi \to \bot$)
- **Outside the trust base**
  - basic theories for equality, sorts, rewriting, etc.
  - K-related lemmas & theorems; ~100,000 lines of proofs
  - a "**database**" of matching logic & K
- An example lemma, (Functional Substitution):

$$\frac{\forall \vec{x}.\, t_{k_1} \wedge p_{k_i} \Rightarrow s_{k_i} \quad \exists y_1.\, \varphi_1 = y_1 \quad \cdots \quad \exists y_m.\, \varphi_m = y_m}{t_{k_i}\theta \ \wedge p_{k_i}\theta \ \Rightarrow s_{k_i}\theta} \quad \begin{array}{l} \theta = [\varphi_1/x_1 \ldots \varphi_m/x_m] \\ y_1, \ldots, y_m \text{ fresh} \end{array}$$

Compiling K into Matching Logic

- How to get Γ^{L}?
  - **Phase 1**: K to Kore (an intermediate); **Phase 2**: Kore to matching logic
- Roughly speaking, Kore = $\Gamma^{equality} + \Gamma^{sorts} + \Gamma^{rewriting}$



Phase 1

Phase 2

Proof Generation

- Our running example
  - $\langle m, n\rangle \Rightarrow \langle m-1, n+m\rangle$ if $m > 0$
  - $\langle 100{,}0\rangle$
    $\Rightarrow \langle 100-1{,}0+100\rangle$
    $= \langle 99{,}0+100\rangle$
    $= \langle 99{,}100\rangle$

rewrite step

simplification/equational step(s)

- **Problem Formulation**
  - semantic/rewrite rules
    $S = \{t_k \wedge p_k \Rightarrow s_k \mid k = 1,2,\dots,K\}$
  - execution trace
    $$\varphi_0, \varphi_1, \dots, \varphi_n$$
  - proof parameter (hint)
    $\Theta = (k_0, \theta_0), \dots, (k_{n-1}, \theta_{n-1})$

$$
\begin{array}{ll}
\Gamma^L \vdash \varphi_0 \Rightarrow s_{k_0}\theta_0 & // \text{ by applying } t_{k_0} \wedge p_{k_0} \Rightarrow s_{k_0} \text{ using } \theta_0 \\
\Gamma^L \vdash s_{k_0}\theta_0 = \varphi_1 & // \text{ by simplifying } s_{k_0}\theta_0 \\
\quad \dots & \\
\Gamma^L \vdash \varphi_{n-1} \Rightarrow s_{k_{n-1}}\theta_{n-1} & // \text{ by applying } t_{k_{n-1}} \wedge p_{k_{n-1}} \Rightarrow s_{k_{n-1}} \text{ using} \\
\Gamma^L \vdash s_{k_{n-1}}\theta_{n-1} = \varphi_n & // \text{ by simplifying } s_{k_{n-1}}\theta_{n-1}
\end{array}
$$

Need (Functional Substitution) to instantiate the rewrite rules

Need $\Gamma^{equality}$ to apply simplification equations

Experiments

- Benchmark
  - REC rewriting competition (http://rec.gforge.inria.fr/)
- Evaluation (2 aspects)
  - Proof generation
  - Proof checking (by Metamath)
- Main takeaway:
  - Fast proof checking (a few seconds)
  - OK proof generation (several minutes)
  - Very large proof objects (millions LOC)

- Proof generation
  - PL semantics Γ^L
  - rewrite steps (linear)
- Proof checking
  - matching logic "database"
  - Proofs for one execution
- Let's see the breakdown analysis

Experiments

Table 1: Performance of proof generation/checking (time measured in seconds).

programs	proof generation			proof checking			proof size	
	sem	rewrite	total	logic	task	total	kLOC	MB
`10.two-counters`	5.95	12.19	18.13	3.26	0.19	3.44	963.8	77
`20.two-counters`	6.31	24.33	30.65	3.41	0.38	3.79	1036.5	83
`50.two-counters`	6.48	73.09	79.57	3.52	0.98	4.50	1259.2	100
`100.two-counters`	6.75	177.55	184.30	3.50	2.10	5.60	1635.6	130
`add8`	11.59	153.34	164.92	3.40	3.09	6.48	1986.8	159
`factorial`	3.84	34.63	38.46	3.57	0.90	4.47	1217.9	97
`fibonacci`	4.50	12.51	17.01	3.44	0.21	3.65	971.7	77
`benchexpr`	8.41	53.22	61.62	3.61	0.80	4.41	1191.3	95
`benchsym`	8.79	47.71	56.50	3.53	0.72	4.25	1163.4	93
`benchtree`	8.80	26.86	35.66	3.47	0.32	3.80	1021.5	81
`langton`	5.26	23.07	28.33	3.46	0.40	3.86	1048.0	84
`mul8`	14.39	279.97	294.36	3.48	7.18	10.66	3499.2	280
`revelt`	4.98	51.83	56.81	3.35	1.10	4.45	1317.4	105
`revnat`	4.81	123.44	128.25	3.37	5.28	8.65	2691.9	215
`tautologyhard`	5.16	400.89	406.05	3.55	14.50	18.04	6884.7	550

matching logic database
(check it once and for all)

Nice performance

Very large proof objects!

Implementation limitations

- Only support the core K features: PL syntax + rewrite rules
  - More complex K features are for future work
- Domain reasoning is assumed
  - No proofs for arithmetic computation, use of SMT solvers, etc.
- From K to matching logic, need Kore
  - Need to trust the K compilation tool (K => Kore)

A <u>Trustworthy</u> Language Framework is Possible

- **Program Execution = Proofs**
  - Correctness justified by proof objects.
- Trust base: 245-LOC code
  - Metamath formalization of matching logic
- Proof objects: very large
  - Proof generation: OK performance
  - Proof checking: very fast
- Why stop at program execution?