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 **program execution** into **mathematical proofs**.
  • 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: $t_0, t_1, t_2, ..., t_n$ (of language L) into a proof $\Gamma^L \vdash t_0 \Rightarrow t_n$
  • Formalize/encode matching logic and “$\Rightarrow$”
  • Translate K formal semantics into $\Gamma^L$
  • Generate the proofs
• Implementation & Experiment
K Overview

- [https://kframework.org](https://kframework.org)
- K = a meta-language to define PLs
  - C, Java, JavaScript, Ethereum VM, Python, Rust, x86-64, etc
- **Language-independence**
  - Proof generation for all languages!
An Example of K

Fig. 2: The complete K formal definition of an imperative language IMP.
Use K to Execute Programs

• Only one rewrite rule:
  \( (m, n) \Rightarrow (m - 1, n + m) \) if \( m > 0 \)

\( (100, 0), (99, 100), (98, 199), \ldots, (1, 5049), (0, 5050) \)

• 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
    $ \textit{kompile} \text{ two-counters.k }$
  • Call K execution tool
    $ \textit{krun} \text{ 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](http://matching-logic.org))
  • K semantics = matching logic theory. E.g., $\Gamma^{\text{two-counters}}$
  • K tools = matching logic proofs. E.g., $\Gamma^{\text{two-counters}} \vdash \langle 100,0 \rangle \Rightarrow \langle 0,5050 \rangle$

• Simple syntax. Simple proof system (next slide)

patterns $\varphi ::= x \mid X \mid \sigma \mid \varphi_1 \varphi_2 \mid \bot \mid \varphi_1 \rightarrow \varphi_2 \mid \exists x. \varphi \mid \mu X. \varphi$

variables, symbols, and application propositional logic quantification fixpoints & induction

• Expressive. Complex concepts defined by axioms/theories
  • theories of equality $\Gamma^{\text{equality}}$, of sorts/types $\Gamma^{\text{sorts}}$, of rewriting $\Gamma^{\text{rewriting}}$
  • theory of a PL $\Gamma^{L} \supseteq \Gamma^{\text{equality}} \cup \Gamma^{\text{sorts}} \cup \Gamma^{\text{rewriting}} \cup \ldots$
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

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](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

![Fig. 6: An extract of the Metamath formalization of matching logic.](image)
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., $\neg \varphi \equiv \varphi \rightarrow \bot$)

• **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):

\[
\forall \bar{x}. t_{k_i} \land p_{k_i} \Rightarrow s_{k_i} \quad \exists y_1. \varphi_1 = y_1 \quad \cdots \quad \exists y_m. \varphi_m = y_m
\]

\[
\theta = [\varphi_1/x_1 \ldots \varphi_m/x_m]
\]

\[
y_1, \ldots, y_m \text{ fresh}
\]

\[
t_{k_i} \theta \land p_{k_i} \theta \Rightarrow s_{k_i} \theta
\]
Compiling K into Matching Logic

• How to get $\Gamma^L$?
  • **Phase 1**: K to Kore (an intermediate); **Phase 2**: Kore to matching logic

• Roughly speaking, Kore = $\Gamma^{equality} + \Gamma^{sorts} + \Gamma^{rewriting}$
Proof Generation

• **Our running example**
  - \(\langle m, n \rangle \Rightarrow \langle m - 1, n + m \rangle\) if \(m > 0\)
  - \(\langle 100, 0 \rangle\)
    - \(\Rightarrow \langle 100 - 1, 0 + 100 \rangle\)
    - \(= \langle 99, 0 + 100 \rangle\)
    - \(= \langle 99, 100 \rangle\)

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

\[ \Gamma^L \vdash \varphi_0 \Rightarrow s_{k_0} \theta_0 \quad \text{by applying } t_{k_0} \land p_{k_0} \Rightarrow s_{k_0} \text{ using } \theta_0 \]
\[ \Gamma^L \vdash s_{k_0} \theta_0 = \varphi_1 \quad \text{by simplifying } s_{k_0} \theta_0 \]
\[ \ldots \]
\[ \Gamma^L \vdash \varphi_{n-1} \Rightarrow s_{k_{n-1}} \theta_{n-1} \quad \text{by applying } t_{k_{n-1}} \land p_{k_{n-1}} \Rightarrow s_{k_{n-1}} \text{ using } \theta_{n-1} \]
\[ \Gamma^L \vdash s_{k_{n-1}} \theta_{n-1} = \varphi_n \quad \text{by simplifying } s_{k_{n-1}} \theta_{n-1} \]

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 \( L \)
  • rewrite steps (linear)

• Proof checking
  • matching logic “database”
  • Proofs for one execution

• Let’s see the breakdown analysis
Table 1: Performance of proof generation/checking (time measured in seconds).

<table>
<thead>
<tr>
<th>programs</th>
<th>proof generation</th>
<th>proof checking</th>
<th>proof size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sem</td>
<td>rewrite</td>
<td>total</td>
</tr>
<tr>
<td>10.two-counters</td>
<td>5.95</td>
<td>12.19</td>
<td>18.13</td>
</tr>
<tr>
<td>20.two-counters</td>
<td>6.31</td>
<td>24.33</td>
<td>30.65</td>
</tr>
<tr>
<td>50.two-counters</td>
<td>6.48</td>
<td>73.09</td>
<td>79.57</td>
</tr>
<tr>
<td>100.two-counters</td>
<td>6.75</td>
<td>177.55</td>
<td>184.30</td>
</tr>
<tr>
<td>add8</td>
<td>11.59</td>
<td>153.34</td>
<td>164.92</td>
</tr>
<tr>
<td>factorial</td>
<td>3.84</td>
<td>34.63</td>
<td>38.46</td>
</tr>
<tr>
<td>fibonacci</td>
<td>4.50</td>
<td>12.51</td>
<td>17.01</td>
</tr>
<tr>
<td>benchexpr</td>
<td>8.41</td>
<td>53.22</td>
<td>61.62</td>
</tr>
<tr>
<td>benchsym</td>
<td>8.79</td>
<td>47.71</td>
<td>56.50</td>
</tr>
<tr>
<td>benchtree</td>
<td>8.80</td>
<td>26.86</td>
<td>35.66</td>
</tr>
<tr>
<td>langton</td>
<td>5.26</td>
<td>23.07</td>
<td>28.33</td>
</tr>
<tr>
<td>mul8</td>
<td>14.39</td>
<td>279.97</td>
<td>294.36</td>
</tr>
<tr>
<td>revelt</td>
<td>4.98</td>
<td>51.83</td>
<td>56.81</td>
</tr>
<tr>
<td>revnat</td>
<td>4.81</td>
<td>123.44</td>
<td>128.25</td>
</tr>
<tr>
<td>tautologyhard</td>
<td>5.16</td>
<td>400.89</td>
<td>406.05</td>
</tr>
</tbody>
</table>

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 **Trustworthy** 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?**