Technical Report:
# Making Formal Verification Trustworthy via Proof Generation

Zhengyao Lin[1], Xiaohong Chen[1], Minh-Thai Trinh[2], John Wang[1], and Grigore Roşu[1]

[1] *University of Illinois at Urbana-Champaign*
{zl38,xc3,jzw2,grosu}@illinois.edu
[2] *Advanced Digital Sciences Center, Illinois at Singapore*
minhthai.t@adsc-create.edu.sg

November 21, 2021

## Abstract

Formal deductive verification aims at proving the correctness of programs via logical deduction. However, the fact that it is usually based on complex program logics makes it error-prone to implement. This paper addresses the important research question of how we can make a deductive verifier *trustworthy* through a *practical* approach. We propose a novel technique to generate *machine-checkable* proof objects to certify each verification task performed by the *language-agnostic* deductive verifier of $\mathbb{K}$—a semantics-based language framework. These proof objects encode formal proofs in matching logic—the logical foundation of $\mathbb{K}$. They have a small 240-line trust base and can be directly verified by third-party proof checkers. Our preliminary experiments show promising performance in generating correctness proofs for deductive verification in different programming languages.

## 1 Introduction

Formal deductive verification [24] is a technique to prove that programs satisfy their formal specifications. While deductive verifiers can guarantee the correctness of programs *in theory*, they are usually based on complex program logics and are themselves error-prone to *implement*.

There has been research to formally verify deductive verifiers and/or verification condition generators using interactive theorem provers [39, 25, 26]. While this approach can provide a strong guarantee for the correctness of a deductive verifier, it requires a considerable amount of effort to mechanize the correctness proofs, which are also specific to one programming language.

In this paper, we aim to tackle the following research question: what is a *practical* and *language-agnostic* approach to make deductive verification *trustworthy*?

To address the language-agnostic part, we base our approach on $\mathbb{K}$, a state-of-the-art semantics-based language framework. $\mathbb{K}$ offers an intuitive meta-language for language designers to define the formal semantics of their programming languages. From such formal semantics, $\mathbb{K}$ automatically generates useful language tools such as parsers, interpreters, model checkers, deductive verifiers [11], program equivalence checkers [29], and many others. In practice, $\mathbb{K}$ has been used to define the complete executable formal semantics of C [14], Java [2], JavaScript [41], Python [16], Ethereum virtual machine (EVM) [23], and x86-64 [12], from which their execution and verification tools are automatically generated. Some of them have evolved into commercial products [17, 35].

However, the current implementation of $\mathbb{K}$ has over 500,000 lines of unverified code in Haskell, Java, and C++, which leads to questions about its trustworthiness. Due to the complexity of $\mathbb{K}$, full formal verification of its codebase is unrealistic. Recent work [3] proposes a technique to generate proof objects that certify each task done by $\mathbb{K}$, but it only supports *concrete program execution* and not verification. So in this paper, we propose novel proof generation techniques for *symbolic execution* and *deductive verification* in $\mathbb{K}$.

Specifically, we reduce the correctness of $\mathbb{K}$'s deductive verification tool to a formal proof in *matching logic*, which is the logical foundation of $\mathbb{K}$ [43, 5]. For each verification task performed by $\mathbb{K}$, we generate a proof object for the following matching logic judgment:

$$\Gamma^L \vdash \varphi_{pre} \Rightarrow_{reach} \varphi_{post} \quad , \qquad \text{where} \tag{1}$$

- $\Gamma^L$ is a matching logic theory (i.e., a set of axioms) that defines the formal semantics of a given language $L$;
- $\vdash$ denotes the matching logic proof system;
- $\varphi_{pre}$ and $\varphi_{post}$ are symbolic formulas that encode the

pre/post-conditions of a verification task;

- $\Rightarrow_{reach}$ denotes *reachability*, which captures the notion of *partial correctness*; that is, any program configuration satisfying $\varphi_{pre}$ either has a finite execution trace reaching one configuration satisfying $\varphi_{post}$, or has an infinite/divergent trace.

The complete proof of Equation (1) is then encoded as a *matching logic proof object* and automatically proof-checked using a third-party proof checker called Metamath [37]. If the proof checking passes, we know that $\mathbb{K}$ is correct in carrying out the verification task specified by Equation (1). This way, we establish the correctness of $\mathbb{K}$'s deductive verifier on a case-by-case basis, via matching logic proof generation and proof checking. As a result, the *trust base* of $\mathbb{K}$'s verification tool is dramatically reduced, from the entire 500,000-line codebase to a 240-line formalization of matching logic.

Our approach is *language-agnostic*. The proof generation algorithm is independent of the actual language semantics (i.e. $\Gamma^L$) and the verification task (i.e. $\varphi_{pre}$ and $\varphi_{post}$). Instead, it is *parametric* in them. Therefore, our approach can be used to generate proof objects that certify deductive verification in all programming languages defined in $\mathbb{K}$.

To evaluate our approach, we implemented our proof generation algorithm and experimented with it on two benchmarks. The first contains arithmetic programs for 3 different languages, which is to demonstrate that our method is language-agnostic. The second benchmark is a selection of C verification examples from the SV-COMP competition [52]. Our experimental results show promising performance in both proof generation and proof checking. For example, it takes 64.2 seconds to generate the proof object for the verification of the `sum` program that calculates the sum from 1 to $n$. The entire proof object (37 megabytes) can be proof-checked in 2.0 seconds on a regular laptop (see Section 5).

To summarize, our ***main contribution*** is a language-agnostic algorithm that generates machine-checkable correctness proofs for the deductive verifier in $\mathbb{K}$. We implemented the following ***main artifacts*** that are submitted with this paper [34]:

- A formalization of deductive verification as reachability formulas (i.e., "$\Rightarrow_{reach}$") in matching logic;
- A proof generation tool for $\mathbb{K}$'s deductive verifier.

We organize the rest of the paper as follows. In Section 2, we give an overview of our approach and introduce the basics of $\mathbb{K}$, reachability logic (for deductive verification), and matching logic. We present our proof generation algorithm in Section 3 and discuss its implementation details and limitations in Section 4. We then show evaluation results of our implementation in Section 5. Finally, we discuss related work in Section 6 and conclude the paper in Section 7.

# 2 Overview and Preliminaries

Our goal is to generate matching logic proof objects as correctness certificates for $\mathbb{K}$'s language-agnostic deductive verifier. To achieve this, our method follows 4 main steps:

1. Given a formal semantics of a language $L$ defined in $\mathbb{K}$, we automatically generate its corresponding matching logic theory $\Gamma^L$.
2. Given a set of *reachability* claims (see Section 2.2):
$$R = \{\varphi_1 \Rightarrow_{reach} \psi_1, \ldots, \varphi_n \Rightarrow_{reach} \psi_n\}$$
we use $\mathbb{K}$'s deductive verifier to prove all claims in $R$.
3. If $\mathbb{K}$ successfully proves all claims, we make it output the *proof hints*, which include all *symbolic execution* steps that $\mathbb{K}$ carries out during verification.
4. From the proof hints, we construct the proof objects for the following matching logic proof goals, encoded in a format that can be checked by Metamath [37]:
$$\Gamma^L \vdash \varphi_1 \Rightarrow_{reach} \psi_1 \quad \ldots \quad \Gamma^L \vdash \varphi_n \Rightarrow_{reach} \psi_n$$

In the above, step 1 is accomplished by using an existing translator that compiles formal semantics in $\mathbb{K}$ into matching logic theories (see [3]), and step 2 is accomplished by using $\mathbb{K}$'s deductive verifier.

Steps 3 and 4 are our main technical contributions. These steps pose two primary challenges:

- $\mathbb{K}$'s deductive verifier is based on *reachability logic* which essentially consists of symbolic execution and coinductive reasoning. However, our goal is to reduce everything to the minimal trust base of matching logic. So we need to *mechanize* the embedding of reachability logic into matching logic, which is theoretically shown to be possible by [5].
- $\mathbb{K}$'s symbolic execution engine supports many complicated features such as evaluation order, conditional rewriting, "otherwise" rules (which are catch-all rules if all other rewrite rules fail to apply), unification modulo axioms, etc. It is a non-trivial task to support the generation of low-level proof objects in matching logic to certify these features.

We discuss how to address these challenges as well as the current limitations of our implementation in Sections 3 and 4.

One advantage of our technique is the encoding of matching logic proof objects using Metamath [37], which is a formal language to encode axioms and proof rules, and construct machine-checkable proofs. By doing so, our proof objects can be verified by all (third-party) Metamath proof checkers [33, 40, 37], which increases the overall trustworthiness of our approach.

Although we use Metamath to encode the proof objects, our main algorithm for proof generation is independent of

```
1  module IMP-SYNTAX
2    imports DOMAINS
3    syntax Exp ::=
4        Int
5      | Id
6      | Exp "+" Exp              [left, strict]
7      | Exp "-" Exp              [left, strict]
8      | "(" Exp ")"                   [bracket]
9    syntax Stmt ::=
10       Id "=" Exp ";"              [strict(2)]
11     | "if" "(" Exp ")" Stmt Stmt  [strict(1)]
12     | "while" "(" Exp ")" Stmt
13     | "{" Stmt "}"                  [bracket]
14     | "{" "}"
15     > Stmt Stmt             [left, strict(1)]
16 endmodule
```

```
17 module IMP
18   imports IMP-SYNTAX
19   syntax KResult ::= Int
20   configuration ⟨ $PGM:Stmt, ·Map ⟩
21   // Variable lookup and assignment
22   rule ⟨ C[X], M ⟩ ⇒ ⟨ C[M(X)], M ⟩
23   rule ⟨ C[X = I], M ⟩ ⇒ ⟨ C[{}], M[X ↦ I] ⟩
24   // Arithmetic expression
25   rule I1 + I2 ⇒ I1 + I2
26   rule I1 - I2 ⇒ I1 − I2
27   // Control flow
28   rule {} S:Stmt ⇒ S
29   rule if (I) S _ ⇒ S requires I ≠ 0
30   rule if (0) _ S ⇒ S
31   rule while (B) S ⇒ if (B) { S while(B) S } {}
32 endmodule
```

Figure 1: The complete formal semantics of an imperative language IMP, defined in $\mathbb{K}$. $X$ is a variable of sort `Id`, $I, I_1, I_2$ are variables of sort `Int`, and $M$ is a variable of sort `Map`. $C$ denotes an *evaluation context*, e.g., $C[\Box] = 1 + \Box$ in which case $C[X + 2] = 1 + (X + 2)$. Rules in lines 25-31 can be applied in any evaluation context.

Metamath. Therefore, we assume the understanding that all matching logic theorems/lemmas presented in this paper have been fully formalized in Metamath and proof-checked as part of our proof objects, and we omit their detailed Metamath encoding. Interested readers can find more details about Metamath in [3, 37] and our encoding of reachability lemmas in our repository [34].

In the following three sections, we give the preliminary background on the main systems used in our work:

1. $\mathbb{K}$ [28]—a language framework where formal language semantics can be defined and language tools can be automatically generated;

2. Reachability logic [44, 10], which is used by $\mathbb{K}$ to perform language-agnostic deductive verification.

3. Matching logic [43, 5], which subsumes reachability logic and serves as the unifying logical foundation of $\mathbb{K}$ in general.

## 2.1 $\mathbb{K}$ Framework

In this section, we use an example to explain how to define formal language semantics in $\mathbb{K}$, and how $\mathbb{K}$ uses the semantics to execute programs.

In Figure 1, we show the complete formal definition of an imperative language IMP in $\mathbb{K}$. The definition includes both syntax (module `IMP-SYNTAX` in the left column) and formal semantics (module `IMP` in the right column). In the syntax module `IMP-SYNTAX`, we define two syntactic categories: `Exp` for arithmetic expressions and `Stmt` for statements. The production rules can have *attributes*, enclosed in square brackets. For example, the `[left]` attribute means left-associativity; the `[strict(2)]` attribute specifies an evaluation order where the second argument is evaluated first.

In the semantic module `IMP`, we first define the *computation configurations* of the language IMP, using the keyword **`configuration`**. A configuration is a term that includes all information about the current state of execution. For IMP, a configuration is a pair consisting of the statement to be executed and a map from variables to their values. Given a program/statement $S$, the initial configuration is $\langle S, \cdot_{Map} \rangle$, where $\cdot_{Map}$ denotes the empty map.

In $\mathbb{K}$, formal semantics is given as a set of *rewrite rules* of the form $lhs \Rightarrow rhs$. $\mathbb{K}$ carries out program execution by repeatedly *matching* the current configuration with the left-hand side of a rewrite rule, and then *rewriting* it to the right-hand side, until no rewrite rules can be matched further, in which case the execution terminates. For example, the following execution step applies the rewrite rule at line 23:

$$\langle \texttt{x = 0; x = 1;}, \cdot_{Map} \rangle \Rightarrow_{exec} \langle \texttt{\{\} x = 1;}, \texttt{x} \mapsto \texttt{0} \rangle$$

where $\Rightarrow_{exec}$ denotes rewriting, i.e., program execution.

From the formal definition of IMP in Figure 1, $\mathbb{K}$ can automatically generate language tools for IMP. In the following, we show how $\mathbb{K}$ performs concrete and symbolic execution.

**Example 1** (Concrete Execution)**.** Consider the program

$$\texttt{SUM}_{10} \equiv \texttt{n = 10; s = 0;}\ \texttt{while (n) \{ s = s + n; n = n - 1; \}}$$

which computes the sum $1 + \cdots + 10$. By matching and applying the rewrite rules of IMP exhaustively, $\mathbb{K}$ generates the following rewriting trace:

$$\langle \texttt{SUM}_{10}, \cdot_{Map} \rangle \Rightarrow_{exec} \langle \texttt{\{\}}, \{\texttt{s} \mapsto 55, \texttt{n} \mapsto 0\} \rangle$$

$$\text{(Consequence)}\quad \frac{\mathcal{T} \models \varphi \to \varphi' \quad A \vdash^{reach}_{C} \varphi' \Rightarrow \psi' \quad \mathcal{T} \models \psi' \to \psi}{A \vdash^{reach}_{C} \varphi \Rightarrow \psi}$$

$$\text{(Axiom)}\quad \frac{\varphi \Rightarrow \psi \in A}{A \vdash^{reach}_{C} \varphi \Rightarrow \psi} \qquad \text{(Abstraction)}\quad \frac{A \vdash^{reach}_{C} \varphi \Rightarrow \psi \quad x \notin FV(\psi)}{A \vdash^{reach}_{C} (\exists x.\, \varphi) \Rightarrow \psi}$$

$$\text{(Reflexivity)}\quad \frac{}{A \vdash^{reach}_{\emptyset} \varphi \Rightarrow \varphi} \qquad \text{(Transitivity)}\quad \frac{A \vdash^{reach}_{C} \varphi \Rightarrow^{+} \varphi' \quad A \cup C \vdash^{reach}_{\emptyset} \varphi' \Rightarrow \psi}{A \vdash^{reach}_{C} \varphi \Rightarrow \psi}$$

$$\text{(Circularity)}\quad \frac{A \vdash^{reach}_{C \cup \{\varphi \Rightarrow \psi\}} \varphi \Rightarrow \psi}{A \vdash^{reach}_{C} \varphi \Rightarrow \psi} \qquad \text{(Case Analysis)}\quad \frac{A \vdash^{reach}_{C} \varphi \Rightarrow \psi \quad A \vdash^{reach}_{C} \varphi' \Rightarrow \psi}{A \vdash^{reach}_{C} \varphi \vee \varphi' \Rightarrow \psi}$$

Figure 2: A sound and relatively complete reachability logic proof system [44]. We omit the *reach* subscript since all formulas involved are reachability formulas. $\Rightarrow^{+}$ denotes reachability after one rewriting step. In (Consequence), $\mathcal{T}$ represents the (canonical) matching logic model of program configurations (see [44]).

**Example 2** (Symbolic Execution)**.** Consider the following program with a *symbolic* integer value $n$:

$$\begin{aligned}\texttt{SUM}(n) \equiv\ & \texttt{n = } n\texttt{; s = 0;} \\ & \texttt{while (n) \{ s = s + n; n = n - 1; \}}\end{aligned} \tag{2}$$

By (symbolically) matching and applying the rewrite rules, $\mathbb{K}$ carries out *symbolic execution*. Unlike concrete execution, symbolic execution creates *branches*. For example, when $\mathbb{K}$ encounters the `while`-loop, it splits the configuration into two branches, based on whether $n$ is zero:

$$\begin{aligned}&(\langle\, \texttt{\{\}}, \{\texttt{s} \mapsto 0, \texttt{n} \mapsto 0\}\,\rangle \wedge n = 0) \vee \\ &(\langle\, \texttt{UNROLLED}, \{\texttt{s} \mapsto 0, \texttt{n} \mapsto n\}\,\rangle \wedge n \neq 0)\end{aligned} \tag{3}$$

where $n = 0$ and $n \neq 0$ are called *path conditions*, and `UNROLLED` is the unfolded loop:

$$\begin{aligned}\texttt{UNROLLED} \equiv\ & \texttt{s = s + n; n = n - 1;} \\ & \texttt{while (n) \{ s = s + n; n = n - 1; \}}\end{aligned}$$

Note that unless we bound the variable $n$, symbolic execution as above does not terminate. Instead, $\mathbb{K}$ generates a growing disjunction of branches with path conditions $n = 0, n - 1 = 0, \ldots, n - k = 0, n - k \neq 0$, for any $k \in \mathbb{N}$.

## 2.2 Deductive Verification using Reachability Logic

Using the same semantics that supports program execution, $\mathbb{K}$ can do deductive verification using a formal calculus called *reachability logic*. In this section, we explain how $\mathbb{K}$'s verification tool proves the functional correctness of `SUM`.

In $\mathbb{K}$, program verification refers to proving *reachability formulas* of the form $\varphi \Rightarrow_{reach} \psi$, where $\varphi, \psi$ are conjunctions of configurations and path conditions such as Equation (3). Intuitively, $\varphi \Rightarrow_{reach} \psi$ states that $\varphi$ rewrites to $\psi$ in finitely many steps or it is divergent (i.e., it has an infinite trace). It is therefore reminiscent of the *partial correctness* interpretation of a Hoare triple [24], except that $\mathbb{K}$ is language-agnostic.

To prove reachability formulas, $\mathbb{K}$ uses two proof techniques: symbolic execution and coinductive reasoning. When symbolic execution does not terminate (e.g. `SUM`), coinduction is used to generalize and prove certain repetitive patterns in the (potentially infinite) rewriting trace.

Embodying these two proof techniques is the *reachability logic* [44], which has a sound and relatively complete proof system shown in Figure 2 to derive *reachability judgments* of the form $A \vdash^{reach}_{C} \varphi \Rightarrow_{reach} \psi$ where $A$ (*axioms*) and $C$ (*circularities*) are two sets of reachability formulas. Axioms are usually semantic rules and circularities are coinduction hypotheses that can only be used after at least one step of rewriting (thus the use of $\Rightarrow^{+}$ in (Transitivity), Figure 2).

Recall the `SUM` example in Equation (2). We can write `SUM`'s function correctness as the following reachability judgment:

$$A \vdash^{reach}_{\emptyset} \langle\, \texttt{SUM}(n), \cdot_{Map}\,\rangle \Rightarrow_{reach} \langle\, \texttt{\{\}}, \{\texttt{s} \mapsto \Sigma_{i=1}^{n} i, \texttt{n} \mapsto 0\}\,\rangle \tag{4}$$

where $A$ is initially the set of semantic rules in IMP.

To prove Equation (4), we first perform symbolic execution on its LHS with a combination of (Axiom), (Transitivity), and (Case Analysis) in Figure 2. This reduces the goal to

$$\langle\, \texttt{LOOP}, \{\texttt{s} \mapsto 0, \texttt{n} \mapsto n\}\,\rangle \Rightarrow_{reach} \langle\, \texttt{\{\}}, \{\texttt{s} \mapsto \Sigma_{i=1}^{n} i, \texttt{n} \mapsto 0\}\,\rangle$$

where $\texttt{LOOP} \equiv \texttt{while (n) \{ s = s + n; n = n - 1; \}}$.

Continuing symbolic execution will lead to an infinite loop, so we instead generalize this goal by introducing a new variable $s$ (using (Consequence) and (Abstraction)):

$$\begin{aligned}A \vdash^{reach}_{\emptyset}\ & \langle\, \texttt{LOOP}, \{\texttt{s} \mapsto s, \texttt{n} \mapsto n\}\,\rangle \Rightarrow_{reach} \\ & \langle\, \texttt{\{\}}, \{\texttt{s} \mapsto s + \Sigma_{i=1}^{n} i, \texttt{n} \mapsto 0\}\,\rangle\end{aligned} \tag{5}$$

Now we can show this holds for any $s$ by coinduction on the (possibly infinite) rewrite trace. We first add Equation (5) to the circularity set $C$ using (Circularity), so we can use it later. Then by symbolically executing the LHS of Equation (5), we obtain two branches:

$$(\langle\, \texttt{\{\}}, \{\texttt{s} \mapsto s, \texttt{n} \mapsto 0\}\,\rangle \wedge n = 0) \vee \tag{6}$$

$$(\langle\, \texttt{UNROLLED}, \{\texttt{s} \mapsto s, \texttt{n} \mapsto n\}\,\rangle \wedge n \neq 0) \tag{7}$$

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) | $\dfrac{\varphi \to \psi}{(\exists x.\, \varphi) \to \psi}\ x \notin FV(\psi)$ |

Fixpoint Rules

| | |
|---|---|
| (Prefixpoint) | $\varphi[(\mu X.\, \varphi)/X] \to \mu X.\, \varphi$ |
| (Knaster-Tarski) | $\dfrac{\varphi[\psi/X] \to \psi}{(\mu X.\, \varphi) \to \psi}$ |

Frame Rules

| | |
|---|---|
| (Propagation$_\bot$) | $C[\bot] \to \bot$ |
| (Propagation$_\vee$) | $C[\varphi \vee \psi] \to C[\varphi] \vee C[\psi]$ |
| (Propagation$_\exists$) | $C[\exists x.\, \varphi] \to \exists x.\, C[\varphi]$ where $x \notin FV(C)$ |
| (Framing) | $\dfrac{\varphi \to \psi}{C[\varphi] \to C[\psi]}$ |

Technical Rules

| | |
|---|---|
| (Existence) | $\exists x.\, x$ |
| (Singleton) | $\neg(C_1[x \wedge \varphi] \wedge C_2[x \wedge \neg\varphi])$ |
| (Substitution) | $\dfrac{\varphi}{\varphi[\psi/X]}$ |

Figure 3: Matching logic proof system (where $C, C_1, C_2$ are *application contexts* which are patterns with a single hole variable $\Box$ such that only pattern applications (i.e. $\varphi_1\ \varphi_2$) appear from the root of the pattern to $\Box$. We denote $C[\varphi] \equiv C[\varphi/\Box]$).

where recall that `UNROLLED` is the loop unrolled once. The first branch Equation (6) is subsumed by the RHS of Equation (5), so we are done with this branch.

For the second branch of Equation (7), we symbolically executed it further to $\langle\, \texttt{LOOP}, \{\texttt{s} \mapsto s + n, \texttt{n} \mapsto n - 1\} \,\rangle \wedge n \neq 0$. Now since we have made at least one rewriting step, we can apply Equation (5) itself on this configuration as a coinduction hypothesis (with substitution $\{s \mapsto s + n, n \mapsto n - 1\}$), and get $\langle\, \{\}, \{\texttt{s} \mapsto s + n + \Sigma_{i=1}^{n-1} i, \texttt{n} \mapsto 0\} \,\rangle \wedge n \neq 0$, which is then subsumed by the RHS of Equation (5). Thus we conclude the proof of Equation (5) and therefore (4).

All parts of this proof can be automated except for coming up with a circularity such as Equation (5). So in $\mathbb{K}$, the user is required to provide this circularity rule for the proof to go through, similar to how one needs to provide a loop invariant in Hoare-style verification.

To conclude, $\mathbb{K}$'s deductive verifier employs a verification logic called reachability logic, which embodies symbolic execution and coinductive reasoning.

In the following section, we show how everything we have introduced so far can be formalized in matching logic, thus leading to our approach of certifying $\mathbb{K}$'s verification tool using proof objects in matching logic.

## 2.3 Matching Logic

Matching logic was proposed in [46] as a means to specify and reason about programs compactly and modularly. It was developed in a series of works [43, 5, 7] and finalized in [4].

Matching logic has been adopted as the logical foundation of $\mathbb{K}$, in the sense that every language definition in $\mathbb{K}$ can be translated to a matching logic theory and all reasoning performed by $\mathbb{K}$ can be reduced to matching logic formal proofs. In this section, we introduce matching logic and show how $\mathbb{K}$'s execution and verification tools in Sections 2.1 and 2.2 can be formalized/embedded into matching logic.

### Matching Logic Syntax and Semantics

We fix two sets of variables $EV$ and $SV$. $EV$ is a set of *element variables*, whose elements are denoted $x$, $y$, ..., while $SV$ is a set of *set variables*, whose elements are denoted $X$, $Y$, ... Matching logic formulas, called *patterns*, are inductively defined as:

**Definition 1.** A *(matching logic) signature* $\Sigma$ is a set of *(constant) symbols*. The set of $\Sigma$-*patterns*, or simply *patterns*, is inductively defined by the following grammar

$$\varphi, \psi \in \mathsf{Pattern} ::= x \in EV \mid X \in SV \mid \sigma \in \Sigma \mid \varphi\ \psi \mid \bot \mid \varphi \to \psi \mid \exists x.\, \varphi \mid \mu X.\, \varphi$$

where the pattern $\varphi\ \psi$ is called an *application*, and for the least fixpoint pattern $\mu X.\, \varphi$, we require that $\varphi$ has no negative occurrences of $X$. Other propositional connectives $\top, \neg, \vee, \wedge$ can be defined as derived constructs as usual. Furthermore, we define $\forall x.\, \varphi \equiv \neg\exists x.\, \neg\varphi$ and $\nu X.\, \varphi \equiv \neg\mu X.\, \neg\varphi[\neg X/X]$.

Intuitively, a pattern is a set of elements that *match* it. For example, $\bot$ is interpreted as the empty set, $\top$ is interpreted as the total set (of any given model), and $\varphi \vee \psi$ (resp. $\varphi \wedge \psi$) is interpreted as the union (resp. intersection) of the interpretations of $\varphi$ and $\psi$. We denote the *free variables* in $\varphi$ by $FV(\varphi)$, and *capture-free substitution* by $\varphi[\psi/x]$ and $\varphi[\psi/X]$.

**Example 3** ($\mathbb{K}$ Configurations)**.** In $\mathbb{K}$, configurations are matching logic patterns. The example in Section 2.1 uses a (constrained) configuration $\langle\, \texttt{SUM}(n), \cdot_{Map} \,\rangle \wedge n \geq 0$, which, from a matching logic point of view, is a conjunction of two patterns. The first pattern is an application of a symbol $\langle\rangle \in \Sigma$ to two arguments: the program $\texttt{SUM}(n)$ and the empty map $\cdot_{Map}$. The second pattern is $n \geq 0$. The resulting conjunction is *matched* by all concrete configurations with $n \geq 0$.

### Proof System

Matching logic has a Hilbert-style proof system developed in previous work [5], shown in Figure 3. The proof system defines the provability relation $\Gamma \vdash \varphi$, which means that there exists a formal proof of $\varphi$ using the proof system. $\Gamma$ is a set of patterns added as additional axioms, which we call a *matching logic theory*.

Matching logic has 15 proof rules falling into 4 categories: FOL rules, frame rules, fixpoint rules, and some miscellaneous rules. For first-order reasoning, matching logic includes the complete proof rules for FOL (see, e.g., [47]). The frame rules enable *frame reasoning*, such as lifting a local implication $\vdash \varphi \to \psi$ to an application context $\vdash C[\varphi] \to C[\psi]$. The fixpoint rules support the standard fixpoint reasoning as in modal $\mu$-calculus [30]. Finally, the 3 miscellaneous rules are needed for some completeness results [5, Theorem 16].

Fixpoint reasoning is particularly important in our work. In matching logic, the least fixpoint pattern $\mu X.\, \varphi$ is interpreted as the smallest set $X$ such that the equation $X = \varphi$ holds ($\varphi$ may include recursive occurrences of $X$), and $\nu X.\, \varphi$ is interpreted as the largest such set. Therefore, the following standard fixpoint reasoning rules are sound [6, Lemma 85]:

$$(\mu\text{-Fixpoint})\ \mu X.\, \varphi \leftrightarrow \varphi[(\mu X.\, \varphi)/X] \qquad (\text{KT})\ \frac{\varphi[\psi/X] \to \psi}{\mu X.\, \varphi \to \psi}$$

$$(\nu\text{-Fixpoint})\ \nu X.\, \varphi \leftrightarrow \varphi[(\nu X.\, \varphi)/X] \qquad (\text{KT}_\nu)\ \frac{\psi \to \varphi[\psi/X]}{\psi \to \nu X.\, \varphi}$$

Intuitively, ($\mu$-Fixpoint) and ($\nu$-Fixpoint) state that $\mu X.\, \varphi$ and $\nu X.\, \varphi$ are indeed *fixpoints*. The (KT) and ($\text{KT}_\nu$) proof rules are a direct logical incarnation of the Knaster-Tarski fixpoint theorem [53] in matching logic, and are what support inductive/coinductive reasoning.

$\mathbb{K}$'s deductive verification tool is based on coinductive reasoning (Section 2.2), which is a special case of fixpoint reasoning. Any coinductive proofs that $\mathbb{K}$ carries out during verification can and should be reduced to the more basic matching logic proof rules such as ($\text{KT}_\nu$). This way, we reduce the complex and error-prone verification algorithms into simpler, machine-checkable matching logic proofs.

### $\mathbb{K}$ Definitions as Matching Logic Theories

The formal definition of a programming language $L$ defined in $\mathbb{K}$ derives a matching logic theory $\Gamma^L$, where the syntax of $L$ is represented by (matching logic) symbols and the semantics is captured by rewrite axioms translated from rewrite rules such as those in Figure 1.

To define rewriting in matching logic, we first define the (one-step) transition relation. Let us introduce a new symbol $\bullet \in \Sigma$, called *one-path next*. Intuitively, for any configuration $\gamma$, the pattern $\bullet\gamma$ is matched by all configurations $\gamma'$ such that $\gamma'$ rewrites to $\gamma$ in one step (i.e., $\gamma'$ satisfies "next" $\gamma$). Then, *one-step rewriting* is defined as follows:

$$\varphi \Rightarrow^1_{exec} \psi \equiv \varphi \to \bullet\psi \qquad // \text{ one-step rewriting}$$

One-step rewriting states that for any $\gamma$ matching $\varphi$, there exists $\gamma'$ matching $\psi$, such that $\gamma$ rewrites to $\gamma'$. Therefore, one-step rewriting captures one-step program execution.

We can define the reflexive and/or transitive closures of one-step rewriting using fixpoint patterns:

$$\Diamond\varphi \equiv \mu X.\, \varphi \vee \bullet X$$
$$\varphi \Rightarrow_{exec} \psi \equiv \varphi \to \Diamond\psi$$
$$\varphi \Rightarrow^+_{exec} \psi \equiv \varphi \to \bullet\Diamond\psi$$

Intuitively, $\Diamond\varphi$ is understood as an infinite disjunction $\varphi \vee \bullet\varphi \vee \bullet\bullet\varphi \ldots$, including all configurations that can reach $\varphi$ in finitely many steps. Hence $\Rightarrow_{exec}$ means zero or more steps of rewriting, and $\Rightarrow^+_{exec}$ means one or more steps of rewriting.

**Example 4** (Concrete/Symbolic Execution)**.** In the `SUM` example in Section 2.1, we explain both concrete and symbolic execution. In matching logic, they can be specified as follows:

$$\Gamma^{\text{IMP}} \vdash \langle\, \texttt{SUM}_{10}, \cdot_{Map} \,\rangle \Rightarrow_{exec} \langle\, \{\}, \{\texttt{s} \mapsto 55, \texttt{n} \mapsto 0\} \,\rangle$$
$$\Gamma^{\text{IMP}} \vdash \langle\, \texttt{SUM}(n), \cdot_{Map} \,\rangle \Rightarrow_{exec} (\langle\, \{\}, \{\texttt{s} \mapsto 0, \texttt{n} \mapsto 0\} \,\rangle \wedge n = 0) \vee (\langle\, \texttt{UNROLLED}, \{\texttt{s} \mapsto 0, \texttt{n} \mapsto n\} \,\rangle \wedge n \neq 0)$$

where $\Gamma^{\text{IMP}}$ is the formal definition of IMP in matching logic.

Reachability extends rewriting by allowing infinite traces:

$$\Diamond_w\varphi \equiv \nu X.\, \varphi \vee \bullet X$$
$$\varphi \Rightarrow_{reach} \psi \equiv \varphi \to \Diamond_w\psi$$
$$\varphi \Rightarrow^+_{reach} \psi \equiv \varphi \to \bullet\Diamond_w\psi$$

where $\Diamond_w\varphi$, called *weak-eventually*, is matched by any configurations that match $\Diamond\varphi$ or are divergent [6, Proposition 115 (20)]. Therefore, reachability captures partial correctness.

**Example 5** (Deductive Verification)**.** The functional correctness of `SUM` in Equation (4) can be specified as:

$$\Gamma^{\text{IMP}} \vdash \langle\, \texttt{SUM}(n), \cdot_{Map} \,\rangle \Rightarrow_{reach} \langle\, \{\}, \{\texttt{s} \mapsto \Sigma_{i=1}^{n} i, \texttt{n} \mapsto 0\} \,\rangle$$

Reachability proof rules (Figure 2) can be derived as theorems using the matching logic proof system (Figure 3) and the above definition of reachability patterns. More specifically, given a reachability judgment $A \vdash^{reach}_{C} \varphi \Rightarrow \psi$, one can encode it as the matching logic pattern

$$\underbrace{\bigwedge_{(\psi_1 \Rightarrow \psi_2) \in A} \Box\left(\forall FV(\psi_1, \psi_2).\, \psi_1 \Rightarrow^+_{reach} \psi_2\right)}_{\text{rules in } A \text{ always hold, and thus we use ``}\Box\text{''}} \wedge \underbrace{\bigwedge_{(\psi_1 \Rightarrow \psi_2) \in C} \circ\Box\left(\forall FV(\psi_1, \psi_2).\, \psi_1 \Rightarrow^+_{reach} \psi_2\right)}_{\text{rules in } C \text{ hold if any step is made, so we use ``}\circ\Box\text{''}} \to \left(\varphi \Rightarrow^{\triangle}_{reach} \psi\right)$$

where $\Rightarrow^{\triangle}$ is $\Rightarrow^{+}$ if $C \neq \emptyset$ and is $\Rightarrow$ otherwise. The operators "$\Box$" and "$\circ$" are defined as

$$\circ\varphi \equiv \neg\bullet\neg\varphi \qquad \text{// "all-path next"}$$
$$\Box\varphi \equiv \nu X.\, \varphi \wedge \circ X \qquad \text{// "always"}$$

In this work, we prove reachability claims in matching logic using the encoding above.

# 3 Certifying $\mathbb{K}$'s Deductive Verifier

First proposed by Ştefănescu *et al.* [11], $\mathbb{K}$ implements a language-agnostic deductive verifier based on *reachability logic*. It is then recently shown in [5] that reachability logic can be embedded into matching logic. This paves the theoretical foundation for our goal, which is to generate matching logic proof objects to certify the correctness of each deductive verification task done by $\mathbb{K}$.

In the following sections, we describe in detail how our technique extracts a formal, machine-checkable matching logic proof from $\mathbb{K}$'s verification algorithm. We begin by discussing the verification algorithm in $\mathbb{K}$, which generalizes and automates the reachability proof rules in Figure 2 (Section 3.1). We then describe the main components of our algorithm, which generate proof objects for symbolic execution (Section 3.2), pattern subsumption (Section 3.3), and finally, coinductive reasoning (Section 3.4).

## 3.1 Overview of Our Goal

In Section 2.2, we have shown an informal reachability proof of the functional correctness of the `SUM` program in Equation (2) whose reachability specification is in Equation (4).

During the proof, we also generalize and prove the following claim using coinduction:

$$A \vdash^{reach}_{\emptyset} \langle\, \texttt{LOOP}, \{\texttt{s} \mapsto s, \texttt{n} \mapsto n\} \,\rangle \Rightarrow_{reach} \langle\, \{\}, \{\texttt{s} \mapsto s + \Sigma_{i=1}^{n} i, \texttt{n} \mapsto 0\} \,\rangle \tag{8}$$

To prove the correctness of `SUM` in $\mathbb{K}$, we write both Equations (4) and (8) in a $\mathbb{K}$ specification and then run $\mathbb{K}$'s verification algorithm to check the correctness of the specification.

The verification algorithm in $\mathbb{K}$ is shown in Algorithm 1. In essence, the algorithm is similar to the proof we give in Section 2.2. $\mathbb{K}$ would take the set of reachability claims $R$ given by the user (containing both the main goal and auxiliary coinduction hypotheses), and for each reachability claim $\varphi_1 \Rightarrow_{reach} \varphi_2 \in R$, $\mathbb{K}$ performs symbolic execution from $\varphi_1$ until all branches are subsumed by the right-hand side $\varphi_2$ (checked by line 7). However, the difference is that after the first step of symbolic execution (line 3), we can apply rules in $R$ as well (line 8), which is justified by coinduction.

Our goal in this paper is to extract a matching logic proof from Algorithm 1. It requires us to generate matching logic proofs for:

- Symbolic execution (e.g. lines 3 and 8);

```
   // Checks the validity of claims in R
   // using symbolic execution and coinduction.
1  procedure checkReachability(R)
2      for φ ⇒_reach ψ ∈ R do
3          Q ← successors(φ) ;
4          if Q = ∅ and Γ^L ⊬ φ → ψ then fail;
5          while Q ≠ ∅ do
6              Pop any φ' from Q ;
7              if Γ^L ⊢ φ' → ψ then continue;
8              Q' ← successors_R(φ') ;
9              if Q' = ∅ then fail;
10             else Q ← Q ∪ Q' ;
```

**Algorithm 1:** Verification algorithm in $\mathbb{K}$ [11]. Here, `successors(`$\varphi$`)` is a set of patterns that are the results of symbolically executing $\varphi$ for one step using the formal semantics (see Section 3.2). `successors`$_I$`(`$\varphi$`)` is the result of applying a rule in $R$ if any one is applicable, otherwise we let `successors`$_R$`(`$\varphi$`)` = `successors(`$\varphi$`)`.

- Pattern subsumption (e.g. lines 4 and 7);
- Coinductive reasoning (e.g. the use of $R$ in line 8).

We discuss how these proofs are generated in detail.

## 3.2 Proof Objects for Symbolic Execution

Let $\Gamma^L$ be the matching logic theory that defines the formal definition of a programming language $L$.

### Problem Formulation

Consider the following $\mathbb{K}$ language definition consisting of $K$ (conditional) rewrite rules:

$$\{lhs_k \wedge q_k \Rightarrow^1_{exec} rhs_k \mid k = 1, 2, \ldots, K\} \subseteq \Gamma^L$$

where $lhs_k$ represents the left-hand side of the rewrite rule, $rhs_k$ represents the right-hand side, and $q_k$ denotes the rewriting condition. For unconditional rules, $q_k$ is $\top$. The notation $\Rightarrow^1_{exec}$ stands for one-step execution, defined in Section 2.3.

In symbolic execution, program configurations often appear with their corresponding *path conditions*. We represent them as $t \wedge p$, where $t$ is a configuration and $p$ is a logical constraint/predicate over the free variables of $t$. We call such patterns *constrained terms*. Constrained terms are matching logic patterns.

Unlike concrete execution, symbolic execution can create *branches*. Therefore, we formulate proof generation for symbolic execution as follows. The *input* is an initial constrained term $t \wedge p$ and a list of final constrained terms $t_1 \wedge p_1, \ldots, t_n \wedge p_n$, which are returned by $\mathbb{K}$ as the result(s) of symbolic executing $t$ under the condition $p$. Each $t_i \wedge p_i$

represents one possible execution trace. Our *goal* is to generate a proof object for the following proof goal:

$$\Gamma^L \vdash t \wedge p \Rightarrow_{exec} (t_1 \wedge p_1) \vee \cdots \vee (t_n \wedge p_n) \qquad \text{(Goal)}$$

In other words, using the notations in Algorithm 1, we need to show the correctness of `successors` by proving $\Gamma^L \vdash \varphi \Rightarrow_{exec}$ `successors(`$\varphi$`)`, which further implies $\Gamma^L \vdash \varphi \Rightarrow_{reach}$ `successors(`$\varphi$`)`.

### Proof Hints

To help generate the proof of (Goal), we instrument $\mathbb{K}$ to output *proof hints*, which include rewriting details, such as which semantic rules are applied and what substitutions are used. Formally, the proof hint for the $j$-th rewrite step consists of:

- a constrained term $t_j^{\text{hint}} \wedge p_j^{\text{hint}}$ before step $j$;
- $l_j$ constrained terms $t_{j,1}^{\text{hint}} \wedge p_{j,1}^{\text{hint}}, .., t_{j,l_j}^{\text{hint}} \wedge p_{j,l_j}^{\text{hint}}$ after step $j$, where for each $1 \leq l \leq l_j$, we also annotate the term with the index $1 \leq k_{j,l} \leq K$ of a rewrite rule and a substitution $\theta_{j,l}$;
- an (optional) constrained term $t_j^{\text{rem}} \wedge p_j^{\text{rem}}$, called the *remainder* of step $j$.

Intuitively, each constrained term $t_{j,l}^{\text{hint}} \wedge p_{j,l}^{\text{hint}}$ represents one execution branch, obtained by applying the $k_{j,l}$-th rewrite rule (i.e., $lhs_{k_{j,l}} \wedge q_{k_{j,l}} \Rightarrow^1_{exec} rhs_{k_{j,l}}$) with substitution $\theta_{j,l}$. The remainder $t_j^{\text{rem}} \wedge p_j^{\text{rem}}$ denotes the branch where no rewrite rules can be applied further and thus the execution gets stuck. Note that $t_j^{\text{hint}}$ and $t_j^{\text{rem}}$ may look different even if no rewrite step is made. This is because the path condition $p_j^{\text{rem}}$ may be stronger than the original condition $p_j^{\text{hint}}$. With this stronger path condidtion, $\mathbb{K}$ may be able to simplify $t_j^{\text{hint}}$ to a different term $t_j^{\text{rem}}$.

From the above proof hint, we can generate a proof for the symbolic execution step. For example, the following is the claim for the $j$-th symbolic execution step:

$$\begin{aligned}\Gamma^L \vdash & \left(t_j^{\text{hint}} \wedge p_j^{\text{hint}}\right) \Rightarrow_{exec} \\ & \left(t_{j,1}^{\text{hint}} \wedge p_{j,1}^{\text{hint}}\right) \vee \ldots \vee \left(t_{j,l_j}^{\text{hint}} \wedge p_{j,l_j}^{\text{hint}}\right) \vee \left(t_j^{\text{rem}} \wedge p_j^{\text{rem}}\right)\end{aligned} \qquad \text{(Step}_j\text{)}$$

Recall that $\Rightarrow_{exec}$ is the *reflexive*-transitive closure of one-step execution, so we can have the remainder configuration at the right-hand side even if no execution is made. To prove (Step$_j$), we need to prove each execution branch: for $1 \leq l \leq l_j$,

$$\Gamma^L \vdash \left(t_j^{\text{hint}} \wedge p_{j,l}^{\text{hint}}\right) \Rightarrow^1_{exec} \left(t_{j,l}^{\text{hint}} \wedge p_{j,l}^{\text{hint}}\right) \qquad \text{(Branch}_{j,l}\text{)}$$

For the remainder branch, we need to prove that:

$$\Gamma^L \vdash \left(t_j^{\text{hint}} \wedge p_j^{\text{rem}}\right) \rightarrow \left(t_j^{\text{rem}} \wedge p_j^{\text{rem}}\right) \qquad \text{(Remainder}_j\text{)}$$

### Proof Generation

Given the above proof hints, we prove (Goal) in three phases:

**Phase 1.** We prove (Branch$_{j,l}$) and (Remainder$_j$) for each step $j$ and branch $1 \leq l \leq l_j$.

**Phase 2.** We combine (Branch$_{j,l}$) and (Remainder$_j$) to obtain a proof of (Step$_j$).

**Phase 3.** We combine (Step$_j$) to prove (Goal).

**Remark 1** (Lemmas and Their Proofs)**.** We need many lemmas about program execution "$\Rightarrow_{exec}$" when we generate the proof objects for symbolic execution. The most important and relevant lemmas are stated explicitly in this paper. In total, 196 new lemmas are formally encoded, and their proofs have been completely worked out based on the 240-line Metamath database of matching logic [3]. These lemmas can be easily reused for future development.

In the following, we explain each proof generation step.

**Phase 1: Proving** (Branch$_{j,l}$) **and** (Remainder$_j$)**.** Recall that (Branch$_{j,l}$) is obtained by applying the $k_{j,l}$-th rewrite rule from the language semantics (where $1 \leq k_{j,l} \leq K$):

$$lhs_{k_{j,l}} \wedge q_{k_{j,l}} \Rightarrow^1_{exec} rhs_{k_{j,l}}$$

According to the proof hint, the corresponding substitution is $\theta_{j,l}$. Therefore, by instantiating the rewrite rule with $\theta_{j,l}$, we obtain the following proof:

$$\Gamma^L \vdash lhs_{k_{j,l}}\theta_{j,l} \wedge q_{k_{j,l}}\theta_{j,l} \Rightarrow^1_{exec} rhs_{k_{j,l}}\theta_{j,l} \qquad (9)$$

Since the condition $q_{k_{j,l}}\theta_{j,l}$ is a predicate on the free variables of Equation (9) and it holds on the left-hand side, it also holds on the right-hand side. Therefore, we prove that:

$$\Gamma^L \vdash lhs_{k_{j,l}}\theta_{j,l} \wedge q_{k_{j,l}}\theta_{j,l} \Rightarrow^1_{exec} rhs_{k_{j,l}}\theta_{j,l} \wedge q_{k_{j,l}}\theta_{j,l} \qquad (10)$$

To proceed with the proof, we need the following lemma:

**Lemma 1** ($\Rightarrow^1_{exec}$ Consequence)**.**

$$\frac{\Gamma^L \vdash \varphi \rightarrow \varphi' \quad \Gamma^L \vdash \varphi' \Rightarrow^1_{exec} \psi' \quad \Gamma^L \vdash \psi' \rightarrow \psi}{\Gamma^L \vdash \varphi \Rightarrow^1_{exec} \psi}$$

Intuitively, Lemma 1 allows us to strengthen the left-hand side and/or weaken the right-hand side of an execution relation. Using Lemma 1, and by comparing our proof goal (Branch$_{j,l}$) with Equation (10), we only need to prove the following two implications, called *subsumptions*:

$$\underbrace{\Gamma^L \vdash \left(t_j^{\text{hint}} \wedge p_{j,l}^{\text{hint}}\right) \rightarrow \left(lhs_{k_{j,l}}\theta_{k_{j,l}} \wedge q_{k_{j,l}}\theta_{k_{j,l}}\right)}_{\text{left-hand side strengthening}}$$

$$\underbrace{\Gamma^L \vdash \left(rhs_{k_{j,l}}\theta_{k_{j,l}} \wedge q_{k_{j,l}}\theta_{k_{j,l}}\right) \rightarrow \left(t_{j,l}^{\text{hint}} \wedge p_{j,l}^{\text{hint}}\right)}_{\text{right-hand side weakening}}$$

These subsumption proofs are common in our proof generation procedure (e.g. (Remainder$_j$) is also a subsumption). We elaborate on subsumption proofs later in Section 3.3.

**Phase 2: Proving** ($\text{Step}_j$)**.** The proof goal ($\text{Step}_j$) is proved by combining the proofs for each branch and the remainder:

$$\Gamma^L \vdash t_j^{\text{hint}} \wedge p_{j,1}^{\text{hint}} \Rightarrow^1_{exec} t_{j,1}^{\text{hint}} \wedge p_{j,1}^{\text{hint}} \qquad (\text{Branch}_{j,1})$$

$$\vdots$$

$$\Gamma^L \vdash t_j^{\text{hint}} \wedge p_{j,l_j}^{\text{hint}} \Rightarrow^1_{exec} t_{j,l_j}^{\text{hint}} \wedge p_{j,l_j}^{\text{hint}} \qquad (\text{Branch}_{j,l_j})$$

$$\Gamma^L \vdash t_j^{\text{hint}} \wedge p_j^{\text{rem}} \rightarrow t_j^{\text{rem}} \wedge p_j^{\text{rem}} \qquad (\text{Remainder}_j)$$

Note that our proof goal ($\text{Step}_j$) uses "$\Rightarrow_{exec}$", while the above use either one-step execution ("$\Rightarrow^1_{exec}$") or implication ("$\rightarrow$"). The following lemma allows us to turn one-step execution and implication (i.e. "zero-step execution") into the reflexive-transitive execution relation "$\Rightarrow_{exec}$":

**Lemma 2** ($\Rightarrow_{exec}$ Introduction)**.**

$$\frac{\Gamma^L \vdash \varphi \rightarrow \psi}{\Gamma^L \vdash \varphi \Rightarrow_{exec} \psi} \qquad \frac{\Gamma^L \vdash \varphi \Rightarrow^1_{exec} \psi}{\Gamma^L \vdash \varphi \Rightarrow_{exec} \psi}$$

Then, we need to verify that the disjunction of all path conditions in the branches (including the remainder) is implied from the initial path condition:

$$\Gamma^L \vdash p_j^{\text{hint}} \rightarrow p_{j,1}^{\text{hint}} \vee \cdots \vee p_{j,l_j}^{\text{hint}} \vee p_j^{\text{rem}} \tag{11}$$

The above implication includes only logical constraints and no configuration terms, and thus involves only *domain reasoning*. Therefore, we simply translate it into an equivalent FOL formula and delegate it to SMT solvers, such as Z3 [13].

From Equation (11), we can prove that the left-hand side of ($\text{Step}_j$), $t_j^{\text{hint}} \wedge p_j^{\text{hint}}$, can be broken down into $l_j + 1$ branches by propositional reasoning:

$$\Gamma^L \vdash \left(t_j^{\text{hint}} \wedge p_j^{\text{hint}}\right) \rightarrow \left(t_j^{\text{hint}} \wedge p_{j,1}^{\text{hint}}\right) \vee \ldots \vee \left(t_j^{\text{hint}} \wedge p_{j,l_j}^{\text{hint}}\right) \vee \left(t_j^{\text{hint}} \wedge p_j^{\text{rem}}\right) \tag{12}$$

Note that that right-hand side of Equation (12) is exactly the disjunction of all the left-hand sides of ($\text{Branch}_{j,l}$) and ($\text{Remainder}_j$). Therefore, to prove the proof goal ($\text{Step}_j$), we use the following lemma, which allows us to combine the executions in different branches into one:

**Lemma 3** ($\Rightarrow_{exec}$ Merge)**.**

$$\frac{\Gamma^L \vdash \varphi_1 \Rightarrow_{exec} \psi_1 \quad \ldots \quad \Gamma^L \vdash \varphi_n \Rightarrow_{exec} \psi_n}{\Gamma^L \vdash \bigvee_{i=1}^{n} \varphi_i \Rightarrow_{exec} \bigvee_{i=1}^{n} \psi_i}$$

**Phase 3: Proving** (Goal)**.** We are now ready to prove our final proof goal for symbolic execution. At a high level, the proof simply uses the reflexivity and transitivity of the program execution relation $\Rightarrow_{exec}$. Therefore, our proof generation method is an iterative procedure. We start with the reflexivity of $\Rightarrow_{exec}$, that is:

$$\Gamma^L \vdash (t \wedge p) \Rightarrow_{exec} (t \wedge p) \tag{13}$$

Then, we repeatedly apply the following steps to *symbolically execute* the right-hand side of Equation (13), until it becomes the same as the right-hand side of (Goal):

1. Suppose we have established a proof of

$$\Gamma^L \vdash (t \wedge p) \Rightarrow_{exec} \left(t_1^{\text{im}} \wedge p_1^{\text{im}}\right) \vee \cdots \vee \left(t_m^{\text{im}} \wedge p_m^{\text{im}}\right) \tag{14}$$

   where $t_1^{\text{im}}$, $p_1^{\text{im}}$, etc. represent the **i**nter**m**ediate configurations and constraints, respectively.

2. Look for a ($\text{Step}_j$) claim of the form

$$\Gamma^L \vdash \left(t_j^{\text{hint}} \wedge p_j^{\text{hint}}\right) \Rightarrow_{exec} \left(t_{j,1}^{\text{hint}} \wedge p_{j,1}^{\text{hint}}\right) \vee \ldots \vee \left(t_{j,l_j}^{\text{hint}} \wedge p_{j,l_j}^{\text{hint}}\right) \vee \left(t_j^{\text{rem}} \wedge p_j^{\text{rem}}\right) \qquad (\text{Step}_j)$$

   such that $t_j^{\text{hint}} \wedge p_j^{\text{hint}} \equiv t_i^{\text{im}} \wedge p_i^{\text{im}}$, for some intermediate constrained term $t_i^{\text{im}} \wedge p_i^{\text{im}}$. Without loss of generality, let us assume that $i = 1$, i.e., it is the first intermediate constrained term $t_1^{\text{im}} \wedge p_1^{\text{im}}$ that can be rewritten/executed using ($\text{Step}_j$).

3. Execute $t_1^{\text{im}} \wedge p_1^{\text{im}}$ in Equation (14) for one step using ($\text{Step}_j$), and obtain the following proof:

$$\Gamma^L \vdash (t \wedge p) \Rightarrow_{exec} \underbrace{\left(t_{j,1}^{\text{hint}} \wedge p_{j,1}^{\text{hint}}\right) \vee \cdots \vee \left(t_{j,l_j}^{\text{hint}} \wedge p_{j,l_j}^{\text{hint}}\right) \vee \left(t_j^{\text{rem}} \wedge p_j^{\text{rem}}\right)}_{\text{right-hand side of } (\text{Step}_j)} \vee \underbrace{\left(t_2^{\text{im}} \wedge p_2^{\text{im}}\right) \vee \ldots \vee \left(t_m^{\text{im}} \wedge p_m^{\text{im}}\right)}_{\text{same as Equation (14)}}$$

Finally, when all symbolic execution steps are applied, we check if the resulting proof goal is the same as (Goal), potentially after permuting the disjuncts on the right-hand side. If yes, then the proof generation method succeeds and we obtain a proof of the goal (Goal). Otherwise, the proof generation method fails, indicating potential mistakes made by $\mathbb{K}$'s symbolic execution engine.

## 3.3 Proof Objects for Pattern Subsumption

It is common to prove the *subsumption* or *implication* of constrained terms in our symbolic execution proof. A subsumption has the form:

$$\Gamma^L \vdash (t \wedge p) \rightarrow (t' \wedge p')$$

We divide it into the following two sub-goals:

$$\Gamma^L \vdash p \rightarrow p' \qquad\qquad \Gamma^L \vdash p \rightarrow (t = t')$$

To prove the first sub-goal $\Gamma^L \vdash p \rightarrow p'$, we note that both $p$ and $p'$ are logical constraints. Therefore, its proof

is delegated to external SMT solvers. To prove the second sub-goal $\Gamma^L \vdash p \to (t = t')$, we first try an SMT solver with all constructors abstracted to uninterpreted functions. If the SMT solver proves the goal with such abstraction, our proof generation method succeeds. Otherwise, we break down $t$ and $t'$ into sub-terms. Specifically, if $t \equiv f(t_1, \ldots, t_n)$ and $t' \equiv f(t'_1, \ldots, t'_n)$, we reduce the sub-goal into a set of goals:

$$\Gamma^L \vdash p \to (t_1 = t'_1) \quad \cdots \quad \Gamma^L \vdash p \to (t_n = t'_n)$$

Then we call our proof generation method recursively on the above sub-goals.

This method is incomplete but covers most simplifications done by $\mathbb{K}$. In general, such subsumption is undecidable since it requires proving first-order theorems in the initial algebra of an equational theory. But some techniques in automated *inductive theorem proving* (such as in Maude ITP [22]) are shown to be effective, and we can adopt them in the future.

### 3.4 Proof Objects for Coinduction

Recall that the verification algorithm in Algorithm 1 performs symbolic execution from the left-hand side of each claim until all branches are subsumed by the right-hand side. While the proof generation procedures in previous sections Sections 3.2 and 3.3 can cover symbolic execution already, the missing part is line 8 in Algorithm 1, where we also apply claims in $R$ itself to perform symbolic execution. This may seem like a circular argument, but the algorithm is in fact performing a coinduction on the rewriting trace. In this section, we describe how we can generate proof objects to justify this coinduction step.

We start with the simplest case when $R$ has only one claim $\varphi \Rightarrow_{reach} \psi$. We assume that we have already rewritten $\varphi$ to some intermediate configuration $\varphi'$ in one or more steps:

$$\varphi \Rightarrow^+_{reach} \varphi' \tag{15}$$

Then if the proof hints indicate that we need to apply the original claim $\varphi \Rightarrow_{reach} \psi$ itself to $\varphi'$ as a coinduction hypothesis, we first generate a proof for this single step:

$$\Box(\forall FV(\varphi, \psi).\, \varphi \Rightarrow_{reach} \psi) \to \varphi' \Rightarrow_{reach} \varphi'' \tag{16}$$

by instantiating the quantifiers $\forall FV(\varphi, \psi)$ using the substitution contained in the proof hints, where $\varphi''$ is the result of applying the claim $\varphi \Rightarrow_{reach} \psi$ as a rewrite rule on $\varphi'$. Recall that this is the encoding of the reachability judgment $\{\varphi \Rightarrow_{reach} \psi\} \vdash^{reach}_{\emptyset} \varphi' \Rightarrow \varphi''$.

Now by (Transitivity) applied on Equations (15) and (16), we can get a proof of

$$\circ\Box(\forall FV(\varphi, \psi).\, \varphi \Rightarrow_{reach} \psi) \to \varphi \Rightarrow^+_{reach} \varphi''$$

which is the encoding of the reachability judgment $\vdash^{reach}_{\{\varphi \Rightarrow_{reach} \psi\}} \varphi \Rightarrow \varphi''$. Then we can continue the symbolic execution of $\varphi''$ using the procedure in Section 3.2 except with an additional circularity premise $\circ\Box(\forall FV(\varphi, \psi).\, \varphi \Rightarrow_{reach} \psi)$.

Finally, if the entire reachability proof can be eventually done, we would have obtained a proof of

$$\circ\Box(\forall FV(\varphi, \psi).\, \varphi \Rightarrow_{reach} \psi) \to \varphi \Rightarrow_{reach} \psi$$

which by (Circularity), gives us $\varphi \Rightarrow_{reach} \psi$ as desired.

In general, we could have $n$ claims in $R = \{\varphi_1 \Rightarrow_{reach} \psi_1, \ldots, \varphi_n \Rightarrow_{reach} \psi_n\}$ and their proofs could arbitrarily invoke each other's coinduction hypothesis. This can be justified by a generalization of (Circularity) [45, Lemma 5]:

$$\text{(Set Circularity)} \quad \frac{A \vdash^{reach}_{R} \varphi \Rightarrow \psi \quad \text{for all } (\varphi \Rightarrow \psi) \in R}{A \vdash^{reach}_{\emptyset} \varphi \Rightarrow \psi \quad \text{for all } (\varphi \Rightarrow \psi) \in R}$$

That is, we can simultaneously add all claims in $R$ as coinduction hypotheses to do a mutual coinduction.

In our current implementation, however, we do not implement the proof of (Set Circularity) in its full generality, and we make the assumption that the proof of each claim would only invoke its own coinduction hypothesis. This is not a restriction in theory, because Roşu *et al.* [45, Lemma 5] has shown that any proof using (Set Circularity) can be reduced to one using only (Circularity).

## 4 Implementation and Limitations

In this section, we discuss some interesting technical details about the implementation of the proof generation algorithms in Section 3, as well as its limitations and future directions.

### 4.1 Our Current Implementation

We implemented the proof generation algorithms in Python. Our implementation takes proof hints as input and generates complete matching logic proof objects, which can be directly proof-checked by any Metamath checker. Besides the algorithms, we also manually encoded and proved 196 new lemmas about rewriting and/or reachability reasoning. These new lemmas are used by the proof generation algorithms. Their formal proof objects (∼4,000 LOC in Metamath) were fully, manually worked out and added to the existing Metamath database of matching logic.

For better performance, we also implemented some optimizations for constructing the proof objects. For example, we used a simple heuristic to cache a proof tree when its size exceeds some given threshold, so we could avoid recomputing the proof of a common proof goal and share the same disk space when storing the proof object.

In terms of memory, we represent proof trees as directed acyclic graphs (DAGs) to share common subtrees. When a lemma is applied and multiple proof trees (DAGs) are combined, we use a greedy algorithm to merge subtrees with the same conclusion.

Even with these optimizations, the proof objects can still be huge (in the order of tens of megabytes). This is primarily due to the space-inefficient text-based encoding of proof objects. To address this issue, we compress the proof objects using generic compression tools such as `xz`[54]. This results usually in over 95% reduction in size. A more detailed evaluation of compression can be found in Section 5.

## 4.2 Limitations and Future Directions

We discuss some important limitations about our current preliminary implementation and how we plan to address them in the future.

### 4.2.1 Need to Trust SMT Solvers

Our current implementation delegates *domain reasoning* to SMT solvers and does not generate proof objects for them. By domain reasoning, we mean $\Gamma^L \vdash \varphi \to \psi$, where $\varphi$ and $\psi$ are logical constraints about domain values, such as integers. To prove such domain properties, we encode them as equivalent FOL formulas and query an SMT solver. This creates a gap in our proof objects, because we need to trust the external SMT solvers.

To remove SMT solvers from the trust base, we need to generate complete proofs for domain reasoning. There has been existing research on generating proof objects for SMT solvers, such as [51, 1], which we can incorporate in our proof generation method.

### 4.2.2 Verifying Nondeterministic Programs

In our current work, we consider only *one-path reachability*. In other words, $\varphi \Rightarrow_{reach} \psi$ holds if $\varphi$ diverges or has *one* finite execution that reaches $\psi$. While one-path reachability is sufficient for the verification of deterministic programs, it does not support nondeterministic or concurrent programs, where we need to verify that *all* finite traces from $\varphi$ can reach $\psi$.

To address the verification of nondeterministic programs, *all-path reachability* was proposed [10]. An all-path reachability $\varphi \Rightarrow^{\forall}_{reach} \psi$ claim holds iff all finite (and maximal) execution traces can reach $\psi$. Therefore, it supports the verification of nondeterministic programs. On deterministic programs, all-path and one-path reachability coincide.

To support all-path reachability, we need to extend our method with the following (Step) axiom, which introduces all-path claims from (one-path) rewrite rules in the semantics $A = \{lhs_1 \Rightarrow rhs_1, \ldots, lhs_K \Rightarrow rhs_K\}$:

$$\text{(Step)} \qquad A \vdash^{reach}_{\emptyset} \varphi \Rightarrow^{\forall}_{reach} (\psi_1 \vee \cdots \vee \psi_K)$$

where for $1 \leq k \leq K$, $\psi_k$ is the result of executing $\varphi$ for one step, using the $k$-th semantic rule $lhs_k \Rightarrow rhs_k$. Intuitively, the (Step) axiom states that an execution step must be made using one of the semantic rules in $A$.

| Task | Gen | Steps | Hint Size | Proof Size | Check |
|---|---|---|---|---|---|
| `sum.imp` | 64 s | 42 | 0.58 MB | 37/1.6 MB | 2.0 s |
| `sum.reg` | 215 s | 108 | 2.24 MB | 111/3.6 MB | 6.5 s |
| `sum.pcf` | 107 s | 22 | 0.29 MB | 38/1.5 MB | 2.2 s |
| `exp.imp` | 61 s | 31 | 0.5 MB | 37/1.5 MB | 1.9 s |
| `exp.reg` | 117 s | 43 | 0.96 MB | 70/2.3 MB | 4.3 s |
| `exp.pcf` | 142 s | 29 | 0.5 MB | 65/2.3 MB | 3.9 s |
| `collatz.imp` | 83 s | 55 | 1.14 MB | 49/1.7 MB | 2.3 s |
| `collatz.reg` | 444 s | 100 | 3.66 MB | 209/4.7 MB | 10.8 s |
| `collatz.pcf` | 217 s | 39 | 1.51 MB | 110/2.2 MB | 6.1 s |
| `product.imp` | 70 s | 42 | 0.62 MB | 44/1.8 MB | 2.7 s |
| `product.reg` | 104 s | 42 | 0.81 MB | 65/2.3 MB | 4.1 s |
| `product.pcf` | 155 s | 48 | 0.82 MB | 80/2.8 MB | 4.5 s |
| `gcd.imp` | 181 s | 93 | 1.9 MB | 74/2.3 MB | 3.7 s |
| `gcd.reg` | 261 s | 73 | 1.92 MB | 124/3.3 MB | 6.7 s |
| `gcd.pcf` | 373 s | 38 | 1.35 MB | 150/3.2 MB | 8.4 s |
| `ln/count-by-1` | 23 s | 25 | 0.24 MB | 28/1.3 MB | 1.6 s |
| `ln/count-by-2` | 25 s | 25 | 0.26 MB | 28/1.3 MB | 1.6 s |
| `ln/count-by-k` | 218 s | 51 | 0.73 MB | 36/1.6 MB | 1.9 s |
| `ln/gauss-sum` | 60 s | 39 | 0.53 MB | 38/1.6 MB | 2.4 s |
| `ln/half` | 76 s | 65 | 1.3 MB | 63/2.2 MB | 3.5 s |
| `ln/nested-1` | 141 s | 84 | 1.88 MB | 104/3.4 MB | 8.8 s |

Figure 4: Performance of our proof generation prototype. From left to right, we list the benchmark, proof **gen**eration time, number of symbolic execution **steps**, proof **hint size**, (uncompressed/compressed) **proof** object **size**, and proof **check**ing time. Tasks with prefix `ln/` are from the `loop-new` benchmark of SV-COMP [52].

### 4.2.3 Handling Existentially Quantified Terms

In $\mathbb{K}$, it is common that the RHS of a reachability claim is *existentially quantified*. For example, if we have a Hoare logic for IMP in Section 2.1, then a Hoare-triple $\{\varphi(x)\}P\{\psi(x)\}$ can be translated to the reachability formula:

$$\langle\, P, \{\mathtt{x} \mapsto x\}\,\rangle \wedge \varphi(x) \Rightarrow_{reach} \exists x'.\, \langle\, \{\}, \{\mathtt{x} \mapsto x'\}\,\rangle \wedge \psi(x')$$

Our current implementation does not support existentially quantified terms as above, which leads to more work in writing specifications. In the case when $x'$ is not definable in terms of $x$, we manually Skolemize the existential quantifiers to uninterpreted functions and constrain these Skolem functions so that the coinduction proof is still valid.

To add support for existential quantifiers, most of the algorithms in Section 3 can stay the same, but we need to extend the subsumption prover in Section 3.3 to support proving implications involving existential quantifications.

# 5 Evaluation

We evaluated our proof generation method using two benchmarks. The first benchmark consists of some verification problems of programs written in three programming languages, which aims at showing that our method is indeed

language-agnostic. The second benchmark is a selection of C verification examples from the SV-COMP competition [52]. We used a machine with Intel i7 processors and 16 GB of RAM. The evaluation results are shown in Figure 4. In the following, we discuss the benchmarks and the evaluation results in detail.

### Benchmarks

To demonstrate that our proof generation method is language-agnostic, we defined three different programming languages in $\mathbb{K}$:

- IMP (see Figure 1): a simple imperative language with a C-like syntax;
- REG: a register-based virtual machine with a fixed number of registers;
- PCF, i.e., *programming computable functions* [42]: a functional language.

We considered the following verification examples:

- `SUM`, which computes $1 + \cdots + n$ for input $n$;
- `EXP`, which computes $n^k$ for inputs $n$ and $k$;
- `COLLATZ`, which computes the Collatz sequence [18] for input $n$ until it reaches 1;
- `PRODUCT`, which computes the product of integers using a loop.
- `GCD`, which computes the greatest common divisor of two integers using the Euclidean algorithm.

These five programs are implemented separately in all three languages (IMP, REG, and PCF). Figure 4 shows that our prototype can handle all the different versions (written in different languages) of these programs without any further effort. The detailed encoding of these verification tasks in $\mathbb{K}$ can be found in our repository [34].

Besides these crafted verification examples, we also considered 6 simple C programs from the `loop-new` benchmark in the SV-COMP competition [52]. It is worth mentioning that our goal in this paper is not to propose any new verification algorithm that can handle hard verification problems. Instead, our goal is to generate machine-checkable proof objects as correctness certificates for verification tasks that are carried out by $\mathbb{K}$.

Furthermore, even with simple arithmetic programs such as `SUM`, the process of symbolic rewriting is complicated, as one can see from the proof object sizes in Figure 4. During the verification of a specification, $\mathbb{K}$ performs many seemingly innocuous operations such as substitution and equational simplification, but to reduce them to matching logic, we have to generate detailed proofs for each small step.

### Evaluation Results

We measured the performance of both proof generation and proof checking. For proof generation, we measured the generation time, number of steps in symbolic execution, proof hint sizes, and proof object sizes. We also measured the compressed proof object sizes using a generic compression tool `xz` [54]. For proof checking, we measured the checking time based on a Rust implementation of Metamath called `smetamath` [40].

The *key highlights* of our evaluation are:

1. Proof checking is fast and takes a few seconds;
2. Proof generation takes longer, often in the order of minutes, depending on the number of symbolic execution steps involved in formal verification;
3. Proof objects are large when encoded in Metamath, but they can be greatly reduced via compression, without sacrificing proof generation/checking time.

We explain the experimental results in detail.

**Proof Generation.** At a high level, proof generation time consists of two phases: (1) the time to generate the formal semantics $\Gamma^L$ from the $\mathbb{K}$ language definitions, and (2) the time to generate the proof objects following the algorithm in Section 3. In our experiments, the first phase is efficient and takes a few seconds, which is roughly linear to the size of the $\mathbb{K}$ language definitions (i.e., the number of formal semantic rules). The second generation phase takes most of the time. It is linear to the number of symbolic execution steps that $\mathbb{K}$ has carried out during verification and the size of intermediate configurations.

Although proof generation takes a significant amount of time, deductive verifiers are *slow in general*, and it takes even more time to infer the right invariants for a verification task. Therefore, we argue that it is acceptable to spend the extra time on generating rigorous and machine-checkable proof objects for deductive verifiers, and establishing the verification results based on a small trust base.

**Proof Checking.** Due to the simplicity of Metamath and the small 240-line formalization of matching logic, proof checking is fast. It is another piece of strong evidence that we *should* generate proof objects for verifiers. Once the proofs are generated, they can be made public as *machine-checkable correctness certificates* of the verification tasks. Anyone concerning about the correctness of the verification can access the public proof objects and check them independently.

**Proof Compression.** As mentioned in Section 4.1, besides some optimizations to reduce the complexity of proofs themselves, we also applied a general compression algorithm using `xz` [54] on the proof objects directly. This leads to a

more than 95% reduction in the proof sizes, as shown in Figure 4. Meanwhile, using incremental compression/decompression has little effect on proof generation and checking time.

To sum up, our proof generation method is language-agnostic and can be used across different languages. The experimental results show that generating proofs as correctness certificates for deductive verifiers is practical and has promising performance. The proof generation time is acceptable while the proof checking time is satisfactory.

# 6 Related Work

There has been a lot of effort in providing formal guarantees for programming language tools, such as deductive verifiers. Generally speaking, there are two approaches. One approach is to formalize and prove the correctness of the *entire* tool. The other approach is to generate correctness certificates for *each run* of the tool, or for each analysis task that it carries out. Clearly, our proof generation method presented in this paper belongs to the second approach.

For the first approach, interactive theorem provers such as Coq [36] and Isabelle/HOL [27] are often used to formalize language tools and prove their correctness. For example, CompCert [32] is a C compiler that is implemented and verified in Coq. CakeML [31] is an implementation of Standard ML [20] and verified in HOL4 [48]. It takes a huge effort in verifying such systems, but when it is done, it gives a strong guarantee for the correctness of the entire tool.

However, we note that the theorem provers used to formalize and verify language tools are themselves intricate and based on complex logical foundations. Therefore, their trust base can be very large. For example, Coq is based on *calculus of inductive constructions* (CIC) [8], which is undoubtedly more complex than matching logic, the logical foundation of our proof generation method. Specifically, the Coq kernel has nearly 25,000 lines of OCaml [9].

In fact, there has been recent research trying to reduce the trust base of theorem provers. For example, Sozeau *et al.* [50, 49] attempted to reduce the trust base of Coq by formalizing the kernel of Coq within Coq. Guneratne *et al.* [15] proposed an alternative type theory to CIC that has a smaller trust base. Harrison *et al.* [21] formalized the OCaml kernel of HOL light within HOL light itself.

The second approach, which is undertaken by our proof generation method, is to generate correctness certificates on a *case-by-case basis*. There have been works to generate proofs for decision procedures in SMT solvers to justify their correctness [51, 1, 38]. Implementations of the LF framework [19] are sometimes used to encode axioms and proofs. To the best of our knowledge, our work is the first to apply this approach to symbolic execution and deductive verification.

# 7 Conclusion

We propose an automated approach to verify the results of formal deductive verification in a language-agnostic manner. We base our approach on $\mathbb{K}$ and its logical foundation, matching logic. For every verification task that $\mathbb{K}$ performs, we generate rigorous and machine-checkable matching logic proof objects, which can be verified by a 240-line proof checker. Our experiment shows promising performance in both proof generation and checking.

# References


[1] Clark Barrett, Leonardo De Moura, and Pascal Fontaine. Proofs in satisfiability modulo theories. *All about proofs, Proofs for all*, 55(1):23–44, 2015.

[2] Denis Bogdănaş and Grigore Roşu. K-Java: A complete semantics of Java. In *Proceedings of the 42$^{nd}$ Symposium on Principles of Programming Languages (POPL'15)*, pages 445–456, Mumbai, India, 2015. ACM.

[3] Xiaohong Chen, Zhengyao Lin, Minh-Thai Trinh, and Grigore Roşu. Towards a trustworthy semantics-based language framework via proof generation. In *Proceedings of the 33$^{rd}$ International Conference on Computer-Aided Verification*, Virtual, July 2021. ACM.

[4] Xiaohong Chen, Dorel Lucanu, and Grigore Roşu. Matching logic explained. *Journal of Logical and Algebraic Methods in Programming*, 120:1–36, 2021.

[5] Xiaohong Chen and Grigore Roşu. Matching $\mu$-logic. In *Proceedings of the 34$^{th}$ Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'19)*, pages 1–13, Vancouver, Canada, 2019. IEEE.

[6] Xiaohong Chen and Grigore Roşu. Matching $\mu$-logic. Technical report, University of Illinois at Urbana-Champaign, 2019.

[7] Xiaohong Chen and Grigore Roşu. A general approach to define binders using matching logic. In *Proceedings of the 25$^{th}$ ACM SIGPLAN International Conference on Functional Programming (ICFP'20)*, pages 1–32, New Jersey, USA, 2020. ACM.

[8] Coq Team. Coq documents: Calculus of inductive constructions. Online at `https://coq.inria.fr/refman/language/cic.html`., 2021.

[9] Coq Team. Coq github repository. `https://github.com/coq/coq`, 2021.

[10] Andrei Ştefănescu, Ştefan Ciobâcă, Radu Mereuţă, Brandon M. Moore, Traian Florin Şerbănuţă, and Grigore Roşu. All-path reachability logic. In *Proceedings*

*of the Joint 25th International Conference on Rewriting Techniques and Applications and 12th International Conference on Typed Lambda Calculi and Applications (RTA-TLCA'14)*, volume 8560, pages 425–440, Vienna, Austria, 2014. Springer.

[11] Andrei Ştefănescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore Roşu. Semantics-based program verifiers for all languages. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16)*, pages 74–91, Amsterdam, Netherlands, 2016. ACM.

[12] Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S. Adve, and Grigore Roşu. A complete formal semantics of x86-64 user-level instruction set architecture. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19)*, pages 1133–1148, Phoenix, Arizona, USA, 2019. ACM.

[13] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, volume 4963, pages 337–340, Budapest, Hungary, 2008. Springer.

[14] Chucky Ellison and Grigore Rosu. An executable formal semantics of C with applications. *ACM SIGPLAN Notices*, 47(1):533–544, 2012.

[15] Ananda Guneratne, Chad Reynolds, and Aaron Stump. Project report: Dependently typed programming with lambda encodings in cedille. In David Van Horn and John Hughes, editors, *Trends in Functional Programming - 17th International Conference, TFP 2016*, volume 10447 of *Lecture Notes in Computer Science*, pages 115–134, College Park, MD, USA, 2016. Springer.

[16] Dwight Guth. A formal semantics of Python 3.3. Technical report, University of Illinois at Urbana-Champaign, 2013.

[17] Dwight Guth, Chris Hathhorn, Manasvi Saxena, and Grigore Roşu. RV-Match: Practical semantics-based program analysis. In *Proceedings of the 28th International Conference on Computer Aided Verification (CAV'16)*, volume 9779, pages 447–453, Toronto, Ontario, Canada, 2016. Springer.

[18] Richard Guy. *Unsolved problems in number theory*, volume 1. Springer Science & Business Media, Berlin, Heidelberg, 2004.

[19] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM (JACM)*, 40(1):143–184, 1993.

[20] Robert Harper, David MacQueen, and Robin Milner. *Standard ML*. Department of Computer Science, University of Edinburgh, Edinburgh, UK, 1986.

[21] John Harrison. Towards self-verification of hol light. In *International Joint Conference on Automated Reasoning*, pages 177–191, Seattle, Washington, USA, 2006. Springer, Springer-Verlag Berlin Heidelberg.

[22] Joseph D Hendrix. *Decision procedures for equationally based reasoning*. PhD thesis, University of Illinois at Urbana-Champaign, 2008.

[23] Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, Brandon Moore, Yi Zhang, Daejun Park, Andrei Ştefănescu, and Grigore Roşu. KEVM: A complete semantics of the Ethereum virtual machine. In *Proceedings of the 2018 IEEE Computer Security Foundations Symposium (CSF'18)*, pages 204–217, Oxford, UK, 2018. IEEE. `http://jellopaper.org`.

[24] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[25] Peter V. Homeier and David F. Martin. A mechanically verified verification condition generator. *The Computer Journal*, 38(2):131–141, 1995.

[26] Marieke Huisman. *Reasoning about Java programs in higher order logic using PVS and Isabelle*. PhD thesis, University of Nijmegen, 2001.

[27] Isabelle Team. Isabelle, 2021. `https://isabelle.in.tum.de/`.

[28] K Team. K framework tools 5.0. `https://github.com/kframework/k`, 2021.

[29] Theodoros Kasampalis, Daejun Park, Zhengyao Lin, Vikram S Adve, and Grigore Roşu. Language-parametric compiler validation with application to llvm. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1004–1019, Virtual, 2021. ACM New York, NY, USA.

[30] Dexter Kozen. Results on the propositional $\mu$-calculus. *Theoretical Computer Science*, 27(3):333–354, 1983.

[31] Ramana Kumar, Magnus O Myreen, Michael Norrish, and Scott Owens. Cakeml: a verified implementation of ml. *ACM SIGPLAN Notices*, 49(1):179–191, 2014.

[32] Xavier Leroy. The CompCert verified compiler, software and commented proof. Available at `https://compcert.org/`, March 2020.

[33] Raph Levien and David A. Wheeler. Metamath verifier in python. `https://github.com/david-a-wheeler/mmverify.py`, 2019.

[34] Lin, Zhengyao and Chen, Xiaohong and Trinh, Minh-Thai and Wang, John and Roşu, Grigore. Matching logic proof checker. https://github.com/kframework/matching-logic-proof-checker, 2021.

[35] Qingzhou Luo, Yi Zhang, Choonghwan Lee, Dongyun Jin, Patrick O'Neil Meredith, Traian Florin Şerbănuţă, and Grigore Roşu. RV-Monitor: Efficient parametric runtime verification with simultaneous properties. In *Proceedings of the 5th International Conference on Runtime Verification (RV'14)*, pages 285–300, Toronto, Canada, 2014. Springer International Publishing.

[36] Coq Team. *The Coq proof assistant*. LogiCal Project, 2021.

[37] Norman Megill and David A. Wheeler. Metamath: a computer language for mathematical proofs. Available at http://us.metamath.org, 2019.

[38] George C Necula and Peter Lee. Proof generation in the touchstone theorem prover. In *Proceedings of the 17th International Conference on Automated Deduction*, pages 25–44, Pittsburgh, Pennsylvania, USA, 2000. Springer, Springer-VerlagBerlin, Heidelberg.

[39] Michael Norrish. C formalised in hol. Technical report, University of Cambridge, Computer Laboratory, 1998.

[40] Stefan O'Rear and Mario Carneiro. Metamath verifier in rust. https://github.com/sorear/smetamath-rs, 2019.

[41] Daejun Park, Andrei Ştefănescu, and Grigore Roşu. KJS: A complete formal semantics of JavaScript. In *Proceedings of the 36th annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*, pages 346–356, Portland, OR, 2015. ACM.

[42] Gordon D. Plotkin. Lcf considered as a programming language. *Theoretical computer science*, 5(3):223–255, 1977.

[43] Grigore Roşu. Matching logic. *Logical Methods in Computer Science*, 13(4):1–61, 2017.

[44] Grigore Roşu, Andrei Ştefănescu, Ştefan Ciobâcă, and Brandon M. Moore. One-path reachability logic. In *Proceedings of the 28th Symposium on Logic in Computer Science (LICS'13)*, pages 358–367, New Orleans, USA, 2013. IEEE.

[45] Grigore Roşu, Andrei Ştefănescu, Ştefan Ciobâcă, and Brandon M. Moore. Reachability logic. Technical Report http://hdl.handle.net/2142/32952, University of Illinois, July 2012.

[46] Grigore Roşu and Wolfram Schulte. Matching logic—extended report. Technical Report Department of Computer Science UIUCDCS-R-2009-3026, University of Illinois at Urbana-Champaign, January 2009.

[47] Joseph R. Shoenfield. *Mathematical logic*. Addison-Wesley Pub. Co, Boston, United States, 1967.

[48] Konrad Slind and Michael Norrish. A brief overview of HOL4. In *International Conference on Theorem Proving in Higher Order Logics*, pages 28–32, Montreal, Canada, 2008. Springer, Springer-Verlag Berlin Heidelberg.

[49] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project. *Journal of Automated Reasoning*, 64:947–999, February 2020.

[50] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq coq correct! verification of type checking and erasure for coq, in coq. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–28, 2019.

[51] Aaron Stump, Duckki Oe, Andrew Reynolds, Liana Hadarean, and Cesare Tinelli. Smt proof checking using a logical framework. *Formal Methods in System Design*, 42(1):91–118, 2013.

[52] SV-COMP. Benchmark for sv-comp. https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks, 2021.

[53] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[54] Tukaani Team. Xz utils. https://tukaani.org/xz/, 2021.