Generating Proof Certificates for a Language-Agnostic Deductive Program Verifier

ZHENGYAO LIN, Carnegie Mellon University, USA
XIAOHONG CHEN, University of Illinois, Urbana-Champaign, USA
MINH-THAI TRINH, Advanced Digital Sciences Center, Illinois at Singapore, Singapore
JOHN WANG, University of Illinois, Urbana-Champaign, USA
GRIGORE ROŞU, University of Illinois, Urbana-Champaign, USA

Previous work on rewriting and reachability logic establishes a vision for a language-agnostic program verifier, which takes three inputs: a program, its formal specification, and the formal semantics of the programming language in which the program is written. The verifier then uses a language-agnostic verification algorithm to prove the program correct with respect to the specification and the formal language semantics. Such a complex verifier can easily have bugs. This paper proposes a method to certify the correctness of each successful verification run by generating a proof certificate. The proof certificate can be checked by a small proof checker. The preliminary experiments apply the method to generate proof certificates for program verification in an imperative language, a functional language, and an assembly language, showing that the proposed method is language-agnostic.

CCS Concepts: • **Theory of computation → Logic and verification**.

Additional Key Words and Phrases: Program Verification, Reachability Logic, Matching Logic

ACM Reference Format:
Zhengyao Lin, Xiaohong Chen, Minh-Thai Trinh, John Wang, and Grigore Roşu. 2023. Generating Proof Certificates for a Language-Agnostic Deductive Program Verifier. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 77 (April 2023), 29 pages. https://doi.org/10.1145/3586029

1 INTRODUCTION

A deductive program verifier proves the correctness of a program with respect to a formal specification. Traditional program verifiers are based on a Hoare-style program logic [Hoare 1969] that is specific to the programming language in question, or on a translation into an intermediate verification language such as Boogie [Barnett et al. 2006]. A *language-agnostic* verifier takes a different approach. It takes as input *both* a program with its formal specification *and* the formal semantics of the programming language in which the program is written, and then uses a language-agnostic verification algorithm to prove the program correct with respect to its specification, using directly the language semantics. Therefore, a language-agnostic verifier supports formal verification of any programming languages, provided that their formal semantics are defined [Ştefănescu et al. 2016].

Authors' addresses: Zhengyao Lin, Computer Science Department, Carnegie Mellon University, USA, zhengyal@cmu.edu; Xiaohong Chen, Department of Computer Science, University of Illinois, Urbana-Champaign, USA, xc3@illinois.edu; Minh-Thai Trinh, Advanced Digital Sciences Center, Illinois at Singapore, Singapore, trinhmt@illinois.edu; John Wang, Department of Computer Science, University of Illinois, Urbana-Champaign, USA, jzw2@illinois.edu; Grigore Roşu, Department of Computer Science, University of Illinois, Urbana-Champaign, USA, grosu@illinois.edu.

2475-1421/2023/4-ART77
https://doi.org/10.1145/3586029

Language-agnostic verification has been implemented in the $\mathbb{K}$ framework (https://kframework.org). $\mathbb{K}$ is a formal language semantics framework that allows language designers to define the formal syntax and semantics of their programming languages. From the formal definition of a programming language, $\mathbb{K}$ automatically generates many language tools, including a parser, an interpreter, a symbolic execution tool, a deductive verifier, and a program equivalence checker [Ştefănescu et al. 2016; Kasampalis et al. 2021]. In practice, $\mathbb{K}$ has been used to formalize many real-world programming languages and virtual machines, including C [Ellison and Rosu 2012], Java [Bogdănaş and Roşu 2015], JavaScript [Park et al. 2015], Python [Guth 2013], Ethereum virtual machine (EVM) [Hildenbrandt et al. 2018], x86-64 [Dasgupta et al. 2019], and LLVM [Li and Gunter 2020]. $\mathbb{K}$ automatically generates implementations and tools for those languages, some of which have become commercial products [Guth et al. 2016; Luo et al. 2014].

Along with the wide application of $\mathbb{K}$ in formalizing and verifying real-world languages and systems arises the concern on the correctness of $\mathbb{K}$ itself. Indeed, the current implementation of $\mathbb{K}$ has over 550,000 lines of unverified code in Haskell, Java, and C++. Internally, $\mathbb{K}$ implements complex data structures, algorithms, translations, and optimizations to guarantee the efficiency of the language implementations and tools that it generates. How do we know that a program that is verified by $\mathbb{K}$ is indeed correct? What is in the trust base of the current implementation of $\mathbb{K}$ when it comes to program verification, and how can we reduce that trust base?

In this paper, we propose a technique to certify the correctness of $\mathbb{K}$'s language-agnostic one-path deductive program verifier. For each successful verification run, we generate a proof certificate in *matching logic*, which is the logical foundation of $\mathbb{K}$ [Chen and Roşu 2019a]. Such a proof certificate includes the entire formal programming language semantics as a set of matching logic axioms and the intended program property as a matching logic formula (called a *pattern*; see Section 3.4). The proof certificate also includes all the detailed proof steps for the program property using a sound matching logic proof system, which has an existing 240-line formalization in Metamath [K Team 2022b]. Specifically, we generate proof certificates for reachability judgments of the following form

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

- Γ^L is the set of matching logic axioms (called a *theory*) generated by $\mathbb{K}$, which defines the formal semantics of a given programming language L;
- $\vdash$ denotes the matching logic proof system shown in Figure 3;
- $\varphi_{pre} \Rightarrow_{reach} \varphi_{post}$ is the matching logic formula/pattern that specifies the program's partial correctness property. φ_{pre} and φ_{post} capture the initial and target computation configurations and $\Rightarrow_{reach}$ denotes the *(one-path) reachability relation*. Intuitively, $\varphi_{pre} \Rightarrow_{reach} \varphi_{post}$ states that for any configuration that satisfies φ_{pre}, either there is a *finite* execution trace that reaches a program configuration satisfying φ_{post}, or the execution is *infinite*.

$\mathbb{K}$ uses a language-agnostic verification algorithm (Algorithm 1) to prove one-path reachability claims. The input of the algorithm consists of the formal semantics Γ^L, the program property $\varphi_{pre} \Rightarrow_{reach} \varphi_{post}$, and some optional user-provided invariants or reachability lemmas. At a high level, the algorithm is based on symbolic execution and coinduction. It symbolically executes φ_{pre} using the formal semantics until it identifies a repetitive behavior (such as a loop that has been unfolded and whose body has been fully executed once). At that time, the algorithm discharges the (repetitive) proof obligation by coinduction. The soundness of such coinductive reasoning is shown in [Roşu et al. 2013] and the algorithm has been been integrated in $\mathbb{K}$.

Our main technical contribution is a set of proof generation procedures (Section 4) that generate the detailed proof steps for reachability judgments such as Equation (1). These proof steps are directly based on the 15-rule matching logic proof system (Figure 3) and are encoded in Metamath [Megill and Wheeler 2019]. Therefore, our proof certificates can be automatically checked by a

Metamath verifier. If the proof certificate for one verification run passes the checking, we know that the language-agnostic verifier of $\mathbb{K}$ is correct on *that* verification run. This way, we establish the correctness of the $\mathbb{K}$ deductive verifier on a case-by-case basis via proof generation and proof checking. As a result, the internal verification algorithm of $\mathbb{K}$—which accounts for about 120,000 lines of Haskell code—is removed from the trusted code base. The detailed matching logic proof steps in the proof certificates explicitly justify each individual verification run of $\mathbb{K}$.

We finished a prototype implementation of our proof generation procedures and evaluated it on two benchmark sets. The first benchmark set contains arithmetic programs written in three programming languages. With this benchmark we demonstrate that our method is for *language-agnostic* verification, working directly with the formal semantics of programming languages. The second benchmark set is a selection of C verification examples from the SV-COMP competition [SV-COMP 2021]. The experimental results show promising performance in both generating and checking the proof certificates. For example, for the verification of the `sum` program that calculates the sum from 1 to n, it takes 105 seconds to generate a proof certificate of 37 megabytes, which is checked within a few seconds on a regular laptop (see Section 6). Our implementation can be found at [Lin et al. 2022].

It should be noted that in this work we only consider proving one execution path correct, known as *one-path reachability reasoning* [Roşu et al. 2013]. Both Algorithm 1 and the proof system in Figure 2 are for proving one-path reachability claims. In this context, the verification is successful if there exists one execution path that satisfies the claim. While one-path reachability reasoning is sufficient for deterministic programs, for concurrent and nondeterministic programs we want to prove that all execution paths are correct, which requires *all-path reachability reasoning* [Ştefănescu et al. 2014]; see also Section 4.1. $\mathbb{K}$ supports both one-path and all-path reachability reasoning but our current work does not support proof generation for all-path reachability reasoning yet; future directions are discussed in Section 7.2.

To summarize, we generate machine-checkable proof certificates for a language-agnostic one-path deductive program verifier in $\mathbb{K}$. For each successful verification, we generate a proof certificate that includes the entire formal semantics as axioms and the program property being verified as a matching logic formula/pattern. The proof certificate also includes the detailed proof steps that formally derive the property from the formal semantics using the sound matching logic proof system. This way, the internal implementation of $\mathbb{K}$'s verification algorithm is justified on a case-by-case basis via machine-checkable proof certificates.

We organize the rest of the paper as follows. In Section 2, we discuss the related work. In Section 3, we give an overview of our approach and introduce the basics of $\mathbb{K}$ and matching logic. We show how to generate proof certificates in Section 4. We discuss our implementation in Section 5 and show the experimental results in Section 6. We discuss the future directions in Section 7 and conclude in Section 8.

2 RELATED WORK

The Two Approaches. There has been a lot of effort in providing formal guarantees for programming language tools such as compilers or deductive verifiers. At a high level, we may identify two approaches. One approach is to formalize and prove the correctness of the *entire* tool. For example, CompCert C [Leroy 2020] is a C compiler that has been formally verified to be exempt from miscompilation issues. The other approach is to generate proof certificates on a case-by-case basis for *each run* of the tool. For example, [Pnueli et al. 1998] presents the translation validation technique to check the result of each compilation against the source program and [Parthasarathy et al. 2021] presents an approach where successful runs of the Boogie verifier are validated using

Isabelle proofs. Our work belongs to the second approach, where proof certificates are generated for each verification task carried out using $\mathbb{K}$.

The first approach tends to yield proofs that are more technically involved and does not work well on an existing tool implementation, and is often conducted on a new implementation that aims at being correct-by-construction from the beginning. However, once it is done, it gives the highest formal guarantee for the correctness of the entire tool, once and for all. Besides CompCert C that we mentioned above, there is also CakeML [Kumar et al. 2014], which is an implementation of Standard ML [Harper et al. 1986] that is formally verified in HOL4 [Slind and Norrish 2008]. In this approach, the proof certificates are often written and proved in an interactive theorem prover such as Coq [Coq Team 2021b] and Isabelle [Isabelle Team 2021], because they provide the expressive power needed for certain correctness results, which are often higher-order, in the sense that they are quantified over all possible programs and/or inputs.

The other "case-by-case" approach generates simpler proof certificates and works better on an existing tool implementation, compared to the above "once-and-for-all" approach. In this approach, the proof certificates only relate the input and output of the language tool in question, without needing to depend on the actual implementation of the tool. For example, the technique of translation validation [Pnueli et al. 1998] checks the correctness of each compilation of an optimized compiler, producing a *verifying* compiler, in contrast to a *verified* compiler such as CompCert C. Recently, the idea was applied to not only compilers but also interpreters and deductive verifiers. For example, [Chen et al. 2021a] generates proof certificates for a language-agnostic interpreter, where each (concrete) execution of a program is certified by a machine-checkable mathematical proof. [Parthasarathy et al. 2021] generates proof certificates for the intermediate verification language (IVL) Boogie, where each transformation from programs to their verification conditions is certified. [Garchery 2021] generates proof certificates for the Why3 verifier [Filliâtre and Paskevich 2013], which is also equipped with an IVL to generate verification conditions. [Wils and Jacobs 2021] generates proof certificates for the VeriFast verifier for C [Jacobs et al. 2011], where each successful verification run is certified with respect to CompCert's Clight big step semantics [Blazy and Leroy 2009]. There have also been works that generate proofs for the decision procedures in SMT solvers to certify their correctness [Barrett et al. 2015; Necula and Lee 2000; Stump et al. 2013].

Both the once-and-for-all and case-by-case approaches provide the same (high) level of correctness guarantee when it comes to one successful run of the tool. Our work follows the case-by-case approach, where proof certificates are generated for each successful verification run of the language-agnostic deductive program verifier of $\mathbb{K}$. Since our proof generation method is parametric in the formal semantics of programming languages, it is language-agnostic.

Trust Base and Proof Checkers. There is an intrinsic distinction between mechanically proving/checking/verifying the correctness of a tool and *trusting* that it is correct. Formal verification *transfers* the trust on the system in question to that on the verifier, which in some cases can be more complex than the system being verified. The system can itself be a verifier, which can then be verified/certified further, following the once-and-for-all or case-by-case approaches above. Most state-of-the-art verified/verifying tools, including ours, involve a large number of nontrivial logical transformations and/or encodings of a formal system into another. In the end, they produce proof certificates that can be automatically checked by a proof checker, which belongs to the *trust base*. The simpler and smaller the proof checker is, the higher trustworthiness we achieve.

Most existing works use a proof assistant such as Coq [Coq Team 2021b] or Isabelle [Isabelle Team 2021] to encode and check the final proof certificates. While proof assistants are commonly used in specifying and reasoning about computer systems, they are complex artifacts. For example, Coq has 200,000 lines of OCaml, and the safety-critical kernel still has 18,000 lines [Coq Team

2021a]. It means that if Coq is used as the final proof checker, there is *at least* 18,000 lines of OCaml code to be trusted. It is difficult for us to find the statistics for other proof assistants and/or theorem provers but we expect they are similar.

Metamath [Megill and Wheeler 2019], on the other hand, is a *tiny* language that can express theorems in abstract mathematics, accompanied by proofs that can be checked by a program, called a Metamath verifier. Internally, the Metamath verifier behaves like an automaton with a stack. Axioms and theorems are associated with unique labels and a proof is a sequence of such labels. To check a proof, one maintains a stack that is empty initially, scans the proof, and pushes/pops the axioms and/or the hypotheses/conclusions of theorems accordingly. If in the end the stack contains exactly one statement that is identical to the theorem being proved, the proof is checked. In particular, it does not need to do any complex inference such as pattern matching or unification, making proof checking very simple. As a result, Metamath has dozens of independently-developed verifiers. [Megill and Wheeler 2019] lists 19 of them, some of which are very small: 550 lines of C#, 400 lines of Haskell, 380 lines of Lua, and 350 lines of Python. As a proof-of-concept, we also implemented a Metamath verifier in 740 lines of Rust [Wang 2022], which supports both regular and compressed proofs, and used it in our experiments (see Section 6).

In our work, we use Metamath to encode the proof certificates. Also, we build on an existing formalization of matching logic and its proof system (Figure 3) in 240 lines of Metamath code [K Team 2022b]. As for what counts as the *actual proof checker* in our approach, there can be different opinions, depending on whether Metamath is regarded as a programming language, or as another calculus whose inference system is implemented in a mainstream language, on top of which the proof system of matching logic is formalized. If Metamath is considered as a programming language, our proof checker has 240 lines. Otherwise, our proof checker consists of the 240-line Metamath definition plus an implementation of Metamath (550 lines of C#, 400 lines of Haskell, etc.), which in total has fewer than 1000 lines.

In our (maybe biased) view, there is no reason to *not* regard Metamath as a programming language like C# and Haskell. Metamath is much simpler than (almost) all programming languages. The fact that Metamath has many independent implementations using different programming languages makes it depend *less* on any particular programming language and its runtime environment, such as compilers and underlying operating systems. Metamath is also bootstrapping, in the sense that the executable of its own verifier (as a piece of machine code run on x86-64 Linux) is formally defined in Metamath itself [Carneiro 2020, Section 6]. What is the highest possible correctness guarantee that we can expect from a proof checker? [Carneiro 2020] proposes five possible levels to which we can prove the correctness of the checker, from the level of a logical rendering of the code to that of the logic gates that make up the computer and even the fabrication process relative to some electrical or physical model (although one may not want to do so because the result will be too specific to that particular computer or digital setup). It is clearly out of the scope of this paper to address all the above questions. The meta-point we want to make here is that proof checking systems such as Metamath have perhaps not received the attention they deserve from the formal verification and theorem proving community.

Finally, we should clarify that the proof checker is not the only code that needs to be trusted with the current implementation of our approach. While we do eliminate the need to trust the verification algorithm, which accounts for about 120,000 lines of Haskell code, we still need to trust that the $\mathbb{K}$ frontend is generating the correct logical encoding of the reachability claims from the user input. We shall discuss the trust base of $\mathbb{K}$ and how our work helps reduce it in Section 7.1 in more detail.

3 OVERVIEW AND PRELIMINARIES

We give an overview of our approach and present the preliminaries on $\mathbb{K}$, the language-agnostic verification algorithm for proving one-path reachability claims, and matching logic, which is the logical foundation of $\mathbb{K}$.

3.1 Overview

There are four main steps to generate proof certificates for formal verification in $\mathbb{K}$.

(1) Given the formal semantics of a programming language L defined in $\mathbb{K}$, we use an existing $\mathbb{K}$ tool called `kompile` (see Figure 5) to compile the semantics into a matching logic theory Γ^L, where the semantic rules for L become axioms in Γ^L.

(2) Given the program property being verified and all the necessary invariants as a set of reachability claims as follows (see Section 3.3):

$$R = \{\varphi_1 \Rightarrow_{reach} \psi_1, \ldots, \varphi_n \Rightarrow_{reach} \psi_n\}$$

we run the $\mathbb{K}$ verifier to verify all claims in R. Since the $\mathbb{K}$ verifier is language-agnostic, it uses directly the formal language semantics Γ^L to verify the claims. The main verification algorithm, as shown in Algorithm 1, carries out symbolic execution and coinductive reasoning to handle repetitive behaviors of the program.

(3) If the verification is successful, $\mathbb{K}$ outputs a *proof hint* that includes all the necessary information for generating the proof certificate, such as the symbolic execution steps that have been carried out. To generate the proof hint, we instrument the $\mathbb{K}$ verifier to obtain its intermediate states during verification and then output the information in a YAML-like syntax.

(4) From the proof hint, we generate concrete proof steps using the matching logic proof system (Figure 3) for all the reachability claims/judgments in R:

$$\Gamma^L \vdash \varphi_1 \Rightarrow_{reach} \psi_1 \quad \ldots \quad \Gamma^L \vdash \varphi_n \Rightarrow_{reach} \psi_n$$

The proofs of these steps constitute our final proof certificate of the correctness of the verification run. This proof certificate is encoded in Metamath [Megill and Wheeler 2019] and can be automatically checked by any Metamath verifier.

Our main technical contribution is (4). These reachability claims are proved by the language-agnostic verification algorithm of $\mathbb{K}$ (Algorithm 1), which implements *one-path reachability reasoning* [Roşu et al. 2013] in Figure 2, which is a special case of *matching logic reasoning* [Chen and Roşu 2019a], using the matching logic proof system in Figure 3. Therefore, our technical contribution is twofold: (a) we generate reachability logic proofs for successful runs of Algorithm 1; and (b) we prove all the reachability logic proof rules using the matching logic proof system. The proofs in (a) are different for each verification runs, while the proofs in (b) are *fixed*, because reachability logic has a fixed set of proof rules. In other words, (b) is a direct but nontrivial mechanization of the theoretical results in [Chen and Roşu 2019a, Section VIII] (see Remark 1). For (a), we further decompose it into three parts: generating proofs for for symbolic execution, for pattern subsumption (i.e., logical implications), and for coinduction. These will be discussed in Section 4. Combining (a) and (b), we reduce Algorithm 1 to rigorous matching logic proofs using the matching logic proof system.

Our proof certificates for $\mathbb{K}$ are encoded in Metamath [Megill and Wheeler 2019], which is a formal language to specify axioms and proof rules and construct machine-checkable proofs. This way, our proof certificates can be directly checked by any third-party Metamath verifier [Levien and Wheeler 2019; Megill and Wheeler 2019; O'Rear and Carneiro 2019]. On the other hand, Metamath is only the format we use to encode proof certificates and is not necessary to understand the main proof generation procedures in Section 4. All the theorems and lemmas in this paper have been

```
module IMP-SYNTAX
  imports DOMAINS
  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)]
endmodule
```

```
module IMP
  imports IMP-SYNTAX
  syntax KResult ::= Int
  configuration ⟨ $PGM:Stmt, ·_Map ⟩
  // Variable lookup and assignment
  rule ⟨ C[X], M ⟩ ⇒ ⟨ C[M(X)], M ⟩
  rule ⟨ C[X = I], M ⟩
       ⇒ ⟨ C[{}], M[X ↦ I] ⟩
  // Arithmetic expression
  rule I1 + I2 ⇒ I1 +_Int I2
  rule I1 - I2 ⇒ I1 −_Int I2
  // Control flow
  rule {} S:Stmt ⇒ S
  rule if (I) S _ ⇒ S requires I ≠ 0
  rule if (0) _ S ⇒ S
  rule while (B) S
       ⇒ if (B) { S while(B) S } {}
endmodule
```

Fig. 1. Complete $\mathbb{K}$ Semantics of IMP (source file `imp.k`). Here, 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 *evaluation contexts*, defined by the strictness attributes.

fully formalized/encoded in Metamath and their detailed proof steps have been completely worked out and proof-checked. Readers can find the complete encoding and derivations of reachability logic proof rules and all the lemmas in [Lin et al. 2022].

In the following, we present the preliminaries on:

(1) The $\mathbb{K}$ formal semantics framework where formal language semantics can be defined and language tools can be automatically generated;
(2) One-path reachability logic [Ştefănescu et al. 2014; Roşu et al. 2013], which powers the language-agnostic deductive verifier in $\mathbb{K}$;
(3) Matching logic [Chen and Roşu 2019a; Roşu 2017], which is a simple logic that serves as the logical foundation of $\mathbb{K}$ and subsumes one-path reachability logic.

3.2 $\mathbb{K}$ Framework

At a high level, $\mathbb{K}$ can be understood as a language for defining programming languages. From the $\mathbb{K}$ definition of a programming language L, all language tools of L are automatically generated by $\mathbb{K}$. In other words, language tools are implemented generically once and for all and then instantiated by a language definition. A typical $\mathbb{K}$ definition of a programming language consists of three main components:

(1) the concrete syntax of the programming language as a BNF grammar;
(2) the computation configurations of the programming language; and
(3) the operational semantics, defined as a set of rewrite rules.

3.2.1 An Example. Figure 1 shows an example $\mathbb{K}$ definition of the folklore language IMP, which is a basic imperative language with a C-style syntax. There are two modules. Module `IMP-SYNTAX` in the left column defines the concrete syntax of IMP while module `IMP` in the right column defines the computation configurations—or simply called configurations—and the rewrite rules. In

`IMP-SYNTAX`, the grammar defines two syntactic categories: `Exp` for arithmetic expressions and `Stmt` for statements, using the conventional BNF grammar. Production rules are separated by "`|`" or "`>`". The latter means that the previous rule has higher priority than the next rule. As an example, the following rule

```
syntax Exp ::= Exp "+" Exp [left, strict]
```

in Figure 1 line 6 defines the syntax of the addition of two arithmetic expressions. $\mathbb{K}$ allows to associate *attributes* with a production rule. The attribute `[left]` means left-associativity, so e_1 + e_2 + e_3 will be parsed as (e_1 + e_2) + e_3. The attribute `[strict]` refers to *strict evaluation* (a.k.a. eager evaluation or call-by-value). It tells $\mathbb{K}$ that to evaluate e_1 + e_2 it must first evaluate both arguments e_1 and e_2 into values, say v_1 and v_2, and then evaluate v_1 + v_2 to v_1 $+_{Int}$ v_2 using the rewrite rule in line 28; here $+_{Int}$ is the built-in arithmetic operation that adds two integers. The strictness attribute `[strict(1)]` for the if-then-else statement in line 13 states that only the first argument (i.e., the condition) should be evaluated. Both then- and else-branches should be frozen and kept unchanged.

In module `IMP`, we define the computation configurations of IMP and its semantic rules. A configuration is a data structure that gathers all the semantic information that is needed for the execution of a program. If we think of $\mathbb{K}$ as an abstract machine that can execute programs of a programming language L, then configurations capture the states of that abstract machine. For IMP, a configuration is simply a pair of an IMP program that is to be executed and a (program) state, which is a mapping from program identifiers/variables to their values, as defined in line 22. Line 22 also specifies the initial configuration $\langle$ `$PGM:Stmt`, $\cdot_{Map}$ $\rangle$, which consists of `$PGM:Stmt`—a special variable that is bound to the program passed to $\mathbb{K}$ for execution—and the empty map $\cdot_{Map}$.

The semantic rules in module `IMP` define a transition system over IMP configurations. For example, line 24 defines the variable-lookup rule

$$\texttt{rule}\ \langle\, C[X], M \,\rangle \Rightarrow \langle\, C[M(X)], M \,\rangle$$

Intuitively, for any configuration where a program identifier X appears within some evaluation context C and the map is M, rewrite X to its value $M(X)$ and keep both the context C and the map M unchanged. In $\mathbb{K}$, evaluation contexts are defined by strictness attributes. In other words, if X is the piece of code to be executed according to the evaluation order determined by the strictness attributes, then apply the above variable-lookup to get its value $M(X)$. Similarly, in lines 25–26, we have the variable-assignment rule

$$\texttt{rule}\ \langle\, C[X\ \texttt{=}\ I], M \,\rangle \Rightarrow \langle\, C[\texttt{\{\}}], M[X \mapsto I] \,\rangle$$

It says that if the assignment statement X `=` I appears within C, rewrite it to the empty statement `{}` and update the map M by assigning X to I.

When the map M is not relevant, we can omit it from the semantic rules. For example, in line 32 we have the if-statement rule

$$\texttt{if}\ (I)\ S\ \_ \Rightarrow S\ \texttt{requires}\ I \neq 0$$

$\mathbb{K}$ will automatically infer and complete the configuration and evaluation context, producing the following equivalent semantic rule:

$$\langle\, C[\texttt{if}\ (I)\ S], M \,\rangle \Rightarrow \langle\, C[S], M \,\rangle\ \texttt{requires}\ I \neq 0$$

3.2.2 $\mathbb{K}$ *Process Overview.* We now explain the process that $\mathbb{K}$ follows to generate the language tools from a language definition. At a high level, the $\mathbb{K}$ process can be divided into the *frontend* phase and the *backend* phase. In the frontend phase, a $\mathbb{K}$ language definition (such as `imp.k` in

Figure 1) is compiled into an intermediate representation called the Kore format by the tool `kompile` (see Section 7.1 for more details). All the backend tools are then based on the Kore representation of the language definition. In this paper, the following two backend tools are relevant:

- `krun`, which supports concrete and symbolic/abstract execution of programs;
- `kprove`, which supports formal verification of program claims.

We explain `krun` in the following and `kprover` in Section 3.3.

Since semantic rules in $\mathbb{K}$ are rewrite rules, program execution means rewriting. To execute a program P, the $\mathbb{K}$ interpreter `krun` firstly constructs an initial configuration for P. Then it looks for a semantic rule whose left-hand side is matched by the configuration and applies the rule. One such match-and-apply cycle accounts for one step of program execution. This process is repeated until no semantic rules are applicable, and then the execution terminates.

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

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

which computes the sum $1 + \cdots + 10$. To execute this program in $\mathbb{K}$, we put it in a source file `sum-10.imp` and pass it to `krun`. Following Figure 1 line 22, `krun` constructs the initial configuration $\langle\, \texttt{SUM}_{10}, \cdot_{Map} \,\rangle$, which contains the program and the empty map. Then, `krun` matches and applies the semantic rules in Figure 1 until termination, generating the following execution path:

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

The final configuration $\langle\, \{\}, \{\texttt{s} \mapsto 55, \texttt{n} \mapsto 0\} \,\rangle$ is the output of `krun`. As expected, $\texttt{SUM}_{10}$ has been fully executed and the value of `s` is 55.

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

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

By applying the semantic rules *symbolically*, $\mathbb{K}$ carries out *symbolic execution*. Unlike concrete execution, symbolic execution creates *branches*. For example, after $\mathbb{K}$ encounters the `while`-loop, it splits the execution into two branches, depending on whether n is zero:

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

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

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

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 $n = 0, n - 1 = 0, \ldots, n - k = 0, n - k \neq 0$, where k is the number of times the loop is unfolded.

3.3 Language-Agnostic Verification using One-Path Reachability Logic

Using the same formal language semantics for program execution, $\mathbb{K}$ supports sound and (relatively) complete deductive verification, using a formal calculus called *one-path reachability logic* [Roşu et al. 2013]. A verification problem is specified by a one-path reachability formula $\varphi \Rightarrow_{reach} \psi$, where φ and ψ are conjunctions of configurations and path conditions such as Equation (3). Intuitively, $\varphi \Rightarrow_{reach} \psi$ states that φ rewrites to ψ on some execution path or it is divergent (i.e., it has an infinite execution path). It is therefore reminiscent of the *partial correctness* interpretation of a Hoare triple [Hoare 1969], except that reachability formulas are language-agnostic and reachability logic reasoning (Figure 2) is based on (and is parametric in) the formal semantics.

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

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

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

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

Fig. 2. One-Path Reachability Logic Proof System. We abbreviate $\Rightarrow_{reach}$ as $\Rightarrow$. In (Consequence), $\mathcal{T}$ denotes the standard configuration model (see [Roşu et al. 2013]), relatively to which the logic is complete.

To prove reachability formulas, $\mathbb{K}$ uses two proof techniques: symbolic execution and coinductive circular reasoning. When symbolic execution does not terminate (e.g. for `SUM`), coinduction is used to generalize and prove certain repetitive patterns in the (potentially infinite) rewriting trace. These two proof techniques are embodied in reachability logic using a sound and relatively complete proof system, shown in Figure 2. The proof system has 7 language-agnostic proof rules that derive reachability judgments of the form $A \vdash_C^{reach} \varphi \Rightarrow_{reach} \psi$ where A (*axioms*) and C (*circularities*) are two sets of reachability formulas. In the beginning, A includes all the semantic rules and C is empty. As the proof proceeds, the current formula being proved can be added to C using (Circularity) and then flushed into A by (Transitivity) after at least one execution step.

For example, the program `SUM` in Equation (2) and its loop invariant can be specified by

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

$$A \vdash_\emptyset^{reach} \langle\, \texttt{LOOP}, \{\texttt{s} \mapsto s, \texttt{n} \mapsto n\} \,\rangle \Rightarrow_{reach} \langle\, \{\}, \{\texttt{s} \mapsto s + \Sigma_{i=1}^{n} i, \texttt{n} \mapsto 0\} \,\rangle \tag{5}$$

where A is the set of all the semantic rules of IMP and `LOOP` $\equiv$ `while (n) { s = s + n; n = n - 1; }`. To prove Equation (4), we first perform symbolic execution on its left-hand side using the proof rules (Axiom), (Transitivity), and (Case Analysis) in Figure 2. Then, the proof goal is reduced to

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

which can be proved by the loop invariant.

To prove the loop invariant eq. (5), we first add it to C using (Circularity) so we can use it later as an axiom. Then, we symbolically execute the left-hand side, which results in a *split* into two execution branches, depending on whether $n = 0$ or not. If $n = 0$, the loop condition fails and the symbolic execution will terminate. We only need to calculate the (symbolic) terminating configuration and prove that it satisfies the right-hand side of the reachability formula. If $n \neq 0$, we symbolically execute the loop body once and obtain $\langle\, \texttt{LOOP}, \{\texttt{s} \mapsto s + n, \texttt{n} \mapsto n - 1\} \,\rangle \wedge n \neq 0$, where `LOOP` shows up again. Since we have made at least one execution, the (Transitivity) proof rule flushes the loop variant from C to A as an axiom, which can then be used to prove the unfolded loop, where s and n are instantiated by $s' = s + n$ and $n' = n - 1$, respectively. Then, we only need to prove the following implication (called *subsumption*) using an SMT solver:

$$\langle\, \{\}, \{\texttt{s} \mapsto s' + \Sigma_{i=1}^{n'} i, \texttt{n} \mapsto 0\} \,\rangle \wedge n' \neq 0 \to \langle\, \{\}, \{\texttt{s} \mapsto s + \Sigma_{i=1}^{n} i, \texttt{n} \mapsto 0\} \,\rangle$$

Thus, we conclude the verification. All the above reasoning has been fully automated in $\mathbb{K}$ except the proposal of the loop invariant, which is provided by the users.

To conclude, $\mathbb{K}$ uses reachability logic to verify reachability properties of programs, using directly the formal semantics of the programming language. At a high level, reachability logic reasoning consists of symbolic execution and coinductive circular reasoning, as formalized by the reachability logic proof system in Figure 2.

3.4 Matching Logic: The Logical Foundation of $\mathbb{K}$

Matching logic was proposed in [Roşu and Schulte 2009] as a means to specify and reason about programs compactly and modularly. It was developed in a series of works [Chen and Roşu 2019a, 2020; Roşu 2017] and finalized in [Chen et al. 2021b]. Matching logic is 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 reasoning. In particular, reachability logic reasoning is a special case of matching logic reasoning [Chen and Roşu 2019a]. In this section, we introduce matching logic and show how program execution and deductive verification (i.e., reachability formulas) can be specified in 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 $x, y, \ldots$, while SV is a set of *set variables*, whose elements are denoted $X, Y, \ldots$ Matching logic formulas, called *patterns*, are inductively defined as follows:

Definition 3.1. A *(matching logic) signature* Σ is a set of *(constant) symbols*. The set of Σ*-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 φ 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 φ and ψ. Application is used to build terms and structures. For example, $f(a, b)$ can be expressed as $((f\ a)\ b)$, where the symbol f is applied to a, and then applied to b, like in functional programming languages. In terms of semantics, an application pattern, just like other patterns, is matched by a set of elements. The least fixpoint pattern $\mu X.\, \varphi$ is the smallest fixpoint (ordered by set inclusion) of φ with respect to X. In other words, it is the smallest solution of the equation $X = \varphi$.

We denote the *free variables* in φ by $FV(\varphi)$, and *capture-free substitution* by $\varphi[\psi/x]$ and $\varphi[\psi/X]$.

Example 3 ($\mathbb{K}$ Configurations)**.** $\mathbb{K}$ configurations can be represented using matching logic patterns. For example, the constrained configuration $\langle\ \mathtt{SUM}(n), \cdot_{Map}\ \rangle \wedge n \geq 0$ from Section 3.2 is a conjunction of two patterns. The first pattern is a term $\langle\ \mathtt{SUM}(n), \cdot_{Map}\ \rangle$, which is the symbol $\langle\rangle \in \Sigma$ applied to the program $\mathtt{SUM}(n)$ and the empty map $\cdot_{Map}$. The second pattern is the logical constraint $n \geq 0$. The resulting conjunction is therefore *matched* by all concrete configurations of the specified structure where the symbolic value $n \geq 0$.

Matching Logic Proof System. Matching logic has a Hilbert-style proof system, shown in Figure 3. The proof system defines the provability relation $\Gamma \vdash \varphi$, which means that there exists a formal proof of φ using the proof system. Γ is a set of patterns added as additional axioms, which we call a *matching logic theory*. All matching logic proof rules fall into 4 categories: FOL reasoning, frame

FOL Rules

(Propositional 1) $\varphi \rightarrow (\psi \rightarrow \varphi)$

(Propositional 2) $(\varphi \rightarrow (\psi \rightarrow \theta)) \rightarrow ((\varphi \rightarrow \psi) \rightarrow (\varphi \rightarrow \theta))$

(Propositional 3) $((\varphi \rightarrow \bot) \rightarrow \bot) \rightarrow \varphi$

(Modus Ponens) $\dfrac{\varphi \quad \varphi \rightarrow \psi}{\psi}$

(∃-Quantifier) $\varphi[y/x] \rightarrow \exists x.\, \varphi$

(∃-Generalization) $\dfrac{\varphi \rightarrow \psi}{(\exists x.\, \varphi) \rightarrow \psi}\ x \notin FV(\psi)$

Fixpoint Rules

(Prefixpoint) $\varphi[(\mu X.\, \varphi)/X] \rightarrow \mu X.\, \varphi$

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

Frame Rules

(Propagation$_\bot$) $C[\bot] \rightarrow \bot$

(Propagation$_\vee$) $C[\varphi \vee \psi] \rightarrow C[\varphi] \vee C[\psi]$

(Propagation$_\exists$) $C[\exists x.\, \varphi] \rightarrow \exists x.\, C[\varphi]$ where $x \notin FV(C)$

(Framing) $\dfrac{\varphi \rightarrow \psi}{C[\varphi] \rightarrow C[\psi]}$

Technical Rules

(Existence) $\exists x.\, x$

(Singleton) $\neg(C_1[x \wedge \varphi] \wedge C_2[x \wedge \neg\varphi])$

(Substitution) $\dfrac{\varphi}{\varphi[\psi/X]}$

Fig. 3. Matching Logic Proof System (where C, C_1, C_2 denote patterns that have a single placeholder variable $\Box$ that appears only within nested symbol applications (and not logical connectives). We denote $C[\varphi] \equiv C[\varphi/\Box]$).

reasoning, fixpoint reasoning, and some technical rules that are needed to certain completeness results (such as [Chen and Roşu 2019a, Theorem 16]). For FOL reasoning, matching logic includes the complete proof rules for FOL (see, e.g., [Shoenfield 1967]). The frame rules enable *frame reasoning*, such as lifting a local implication $\vdash \varphi \rightarrow \psi$ to an application context $\vdash C[\varphi] \rightarrow C[\psi]$. The fixpoint rules support the standard fixpoint reasoning as in modal μ-calculus [Kozen 1983].

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 (φ 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 [Chen and Roşu 2019b, Lemma 85]:

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

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

Intuitively, (μ-Fixpoint) and (ν-Fixpoint) state that $\mu X.\, \varphi$ and $\nu X.\, \varphi$ are indeed *fixpoints*. The (KT) and (KT_ν) proof rules are a direct logical incarnation of the Knaster-Tarski fixpoint theorem [Tarski 1955] in matching logic, making inductive/coinductive reasoning sound. The coinductive reasoning used by the $\mathbb{K}$ deductive verifier (Section 3.3), for example, 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 (KT_ν). 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 $\mathbb{K}$ definition of a programming language L derives a matching logic theory Γ^L, where the syntax of L is represented by matching logic symbols and the semantics is captured by axioms translated from the semantics rules such as those in Figure 1. To define semantic/rewrite rules, 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 γ, the pattern $\bullet\gamma$ is matched by all configurations γ' such that γ' rewrites to γ in one step (i.e., γ' satisfies "next" γ). Then, *one-step rewriting* is defined using the following pattern:

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

One-step rewriting states that for any γ matching φ, there exists γ' matching ψ, such that γ rewrites to γ'. Therefore, one-step rewriting captures one-step program execution. Then, we can define the

reflexive and/or transitive closures of one-step rewriting using fixpoints:

$$\begin{aligned} \Diamond\varphi &\equiv \mu X.\, \varphi \vee \bullet X && \text{// “eventually”} \\ \varphi \Rightarrow_{exec} \psi &\equiv \varphi \rightarrow \Diamond\psi && \text{// “rewriting”} \\ \varphi \Rightarrow^{+}_{exec} \psi &\equiv \varphi \rightarrow \bullet\Diamond\psi && \text{// “rewriting (at least one step)”} \end{aligned}$$

Intuitively, $\Diamond\varphi$ is matched by all the configurations that can reach φ 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 3.2, we explain both concrete and symbolic execution. In matching logic, they are formalized as follows:

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

where Γ^{IMP} is the formal definition of IMP in matching logic and n is a free element variable.

Formal deductive verification is specified using reachability relations, which extends the rewriting relations by allowing infinite execution paths:

$$\begin{aligned} \Diamond_w\varphi &\equiv \nu X.\, \varphi \vee \bullet X && \text{// “weak-eventually”} \\ \varphi \Rightarrow_{reach} \psi &\equiv \varphi \rightarrow \Diamond_w\psi && \text{// “reachability”} \\ \varphi \Rightarrow^{+}_{reach} \psi &\equiv \varphi \rightarrow \bullet\Diamond_w\psi && \text{// “reachability (at least one step)”} \end{aligned}$$

where $\Diamond_w\varphi$, called *weak-eventually*, is matched by any configurations that match $\Diamond\varphi$ or are divergent [Chen and Roşu 2019b, Proposition 115 (20)]. This encoding captures partial correctness.

Example 5 (Deductive Verification)**.** The correctness of `SUM` in Equation (4) is formalized as:

$$\Gamma^{\text{IMP}} \vdash \langle\, \texttt{SUM}(n), \cdot_{Map} \,\rangle \Rightarrow_{reach} \langle\, \{\}, \{\texttt{s} \mapsto n(n+1)/2, \texttt{n} \mapsto 0\} \,\rangle$$

Reachability proof rules (Figure 2) can be derived using the matching logic proof system (Figure 3). In other words, they are derived theorems in matching logic. More specifically, a reachability judgment $A \vdash^{reach}_{C} \varphi \Rightarrow \psi$ is encoded as the following pattern [Chen and Roşu 2019a, Section VIII]:

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

where $\Rightarrow^{\triangle}$ is $\Rightarrow^{+}$ if $C \neq \emptyset$ and $\Rightarrow$ otherwise. Intuitively, it means that to move the circularities in C to the axiom set A, we need to make at least one step using the semantics. The operators “$\Box$” and “$\circ$” are defined in the usual way:

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

In this work, we use the above matching logic encoding of one-path reachability claims.

4 GENERATING PROOF CERTIFICATES FOR $\mathbb{K}$'S VERIFICATION TOOL

In this section, we describe in detail how to generate matching logic proof certificates for the language-agnostic program verifier in $\mathbb{K}$. We first review the verification algorithm (Algorithm 1) that automates the reachability proof rules in Figure 2. Then, we describe the main procedures for proof certificate generation, including those for symbolic execution (Section 4.2), pattern subsumption (Section 4.3), and coinductive reasoning (Section 4.4).

```
1  procedure proveAllClaims(R)
2      foreach φ ⇒_reach φ′ ∈ R do
3          if proveOneClaim(R, φ ⇒_reach φ′) = failure then return failure;
4      return success;
5  // a nondeterministic algorithm for proving one reachability claim
6  procedure proveOneClaim(R, φ ⇒_reach φ′)
7      if Γ^L ⊢ φ → φ′ then return success;
8      Q := successors(φ);
9      while Q ≠ ∅ do
10         ψ_front := choose(Q);    // a nondeterministic choice
11         if Γ^L ⊢ ψ_front → φ′ then return success;
12         else Q := successors_R(ψ_front);
13     return failure;
```

Algorithm 1: Algorithm for proving one-path reachability claims. The input R is a set of reachability claims that are to be proved altogether. `proveAllClaims` calls `proveOneClaim` on every claim in R. `proveOneClaim` is presented as a nondeterministic algorithm with a nondeterministic "`choose`" operator at line 10. The verification is successful if there exists one successful run of the algorithm. Both `successors` (line 8) and $\texttt{successors}_R$ (line 12) calculate all the successors of a given configuration. `successors` uses only the formal semantics in Γ^L while $\texttt{successors}_R$ uses both the semantic rules and the claims in R. This is sound because at least one real semantic step has been made in line 8; see Appendix A for details about the soundness proof (provided as supplemental material to this paper). One-path reachability logic reasoning is implemented in $\mathbb{K}$ but is not published. To make the paper self-contained we re-present the algorithm and its soundness proof.

4.1 Overview of the $\mathbb{K}$ Verification Algorithm

We show the language-agnostic verification algorithm of $\mathbb{K}$ in Algorithm 1, which is an optimized implementation of the reachability proof rules in Figure 2. The input R is a set of reachability claims to be verified, including the necessary invariant claims. The algorithm consists of two procedures: `proveAllClaims` and `proveOneClaim`. The first calls the latter on every input claim. The procedure `proveOneClaim` starts by checking the subsumption $\Gamma^L \vdash \varphi \to \varphi'$. If it holds, then the claim $\varphi \Rightarrow_{reach} \varphi'$ is trivially true. If the direct subsumption is false, we perform symbolic execution for one step from φ to get a set Q of all its successors. If $Q \neq \emptyset$, the algorithm *nondeterministically* chooses a frontier pattern ψ_{front} from Q and checks whether ψ_{front} satisfies φ'. If yes, the verification succeeds (line 11). Otherwise, the algorithm symbolically executes ψ_{front} and continues with its successors (line 12), following both the semantic rules and the claims in R. This is sound because in line 8, before the `while` loop, we have computed the successors of φ using only the semantic rules. Immediately after that, when we entered the loop for the first time, we chose one successor of φ, say φ_s (line 10). Therefore, we have $\Gamma^L \vdash \varphi \Rightarrow^+_{reach} \varphi_s$. Since at least one execution step has been made, the (Transitivity) rule in Figure 2 moves all the circularity claims (i.e., the claims in R) to the axiom set so they can be used as semantic axioms in computing further successors (line 12). See Appendix A for the soundness proof (provided as supplemental material to this paper).

In this work we only consider verifying reachability claims on one path, known as *one-path reachability* [Roşu et al. 2013]. The procedure `proveOneClaim` nondeterministically chooses a frontier pattern ψ_{front} from all the possible successors in Q (see line 10), which amounts to looking for the one

execution path that satisfies the reachability claim. Therefore, `proveOneClaim` is successful if there exists a successful run, in which case a particular execution trace is found as the witness of the claim being verified. Based on this execution trace, we can generate a matching logic proof certificate. On the other hand, `proveOneClaim` fails if there is no successful run. A deterministic implementation of `proveOneClaim` will require backtracking for all the nondeterministic choice(s) in line 10. In this work we consider proof generation for *successful* verification runs so we always assume that there is a successful run of line 10. Finally, the procedure `proveAllClaims` calls `proveOneClaim` on all claims in R and the entire verification is successful if `proveAllClaims` is successful.

Before we get into the technical detail of proof generation, we explain the difference between verifying one-path and all-path reachability claims. As we have seen, a one-path reachability claim $\varphi \Rightarrow_{reach} \varphi'$ states the existence of one execution path that satisfies φ' (or is divergent, due to the partial-correctness semantics). However, for concurrent and nondeterministic programs, we often want to verify that *all* execution paths satisfy φ', which motivates all-path reachability logic [Ştefănescu et al. 2014, 2016]. The verification of all-path reachability claims is supported by a modified version of Algorithm 1 where the nondeterministic choice in line 10 is eliminated and *all* execution paths are checked (see Appendix B in the supplemental material for a detailed comparison). In terms of proof systems, all-path reachability logic extends one-path reachability logic with an extra axiom called (Step) (see Section 7.2). The (Step) axiom derives all-path reachability claims from the (one-path) semantic rules in Γ^L and thus serves as the basis of verifying all-path claims. However, the current $\mathbb{K}$ pipeline that translates $\mathbb{K}$ into matching logic is incomplete and the matching logic theory Γ^L does not have the (Step) axiom. Thus in this work, we only consider proof generation for one-path reachability reasoning.

Our goal is to generate matching logic proof certificates for Algorithm 1. For clarity, we divide it into three proof generation procedures:

- Generating proofs for symbolic execution (corresponding to lines 8 and 12);
- Generating proofs for pattern subsumption (corresponding to line 11);
- Generating proofs for coinductive reasoning (corresponding to the use of R in line 12).

We discuss these proof generation procedures in the following.

4.2 Generating Proofs for Symbolic Execution

We use Γ^L to denote the matching logic theory of the formal semantics of a language L.

Problem Formulation. Consider the following $\mathbb{K}$ language definition, which consists 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. Unconditional rules can be regarded as conditional rules where q_k is $\top$. The notation $\Rightarrow^1_{exec}$ stands for one-step execution, defined in Section 3.4.

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 for the following goal:

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

In other words, here we are certifying the correctness of the `successors` (and $\texttt{successors}_R$) methods used by Algorithm 1, by proving that $\Gamma^L \vdash \varphi \Rightarrow_{exec} \texttt{successors}(\varphi)$, which further implies $\Gamma^L \vdash \varphi \Rightarrow_{reach} \texttt{successors}(\varphi)$.

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

- a constrained term $t_j^{\text{hint}} \wedge p_j^{\text{hint}}$ that represents the configuration before step j;
- l_j constrained terms $t_{j,1}^{\text{hint}} \wedge p_{j,1}^{\text{hint}}, .., t_{j,l_j}^{\text{hint}} \wedge p_{j,l_j}^{\text{hint}}$ that represent the configurations after step j, where for each $1 \leq l \leq l_j$, we also annotate it with an index $1 \leq k_{j,l} \leq K$ that refers to the $k_{j,l}$-th semantic rule in Γ^L and a substitution $\theta_{j,l}$;
- an (optional) constrained term $t_j^{\text{rem}} \wedge p_j^{\text{rem}}$, where $p_j^{\text{rem}} \equiv p_j^{\text{hint}} \wedge \neg \left(p_{j,1}^{\text{hint}} \vee \cdots \vee p_{j,l_j}^{\text{hint}}\right)$, called the *remainder* of step j, representing the part/fragment of the original configuration that "gets stuck".

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 semantic rule (i.e., $lhs_{k_{j,l}} \wedge q_{k_{j,l}} \Rightarrow^1_{exec} rhs_{k_{j,l}}$) using substitution $\theta_{j,l}$. The remainder $t_j^{\text{rem}} \wedge p_j^{\text{rem}}$ denotes the branch where no semantic rules can be applied further and thus the execution gets stuck. Note that t_j^{hint} and t_j^{rem} may not be syntactically identical, even if no execution has been made. This is because the path condition p_j^{rem} is stronger than the original condition p_j^{hint}. With this stronger path condition, $\mathbb{K}$ can simplify t_j^{hint} further to t_j^{rem}.

From the above proof hint, we can generate the proof for one symbolic execution step. For example, the following specifies the j-th symbolic execution step:

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

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

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

And for the remainder branch, we need to prove

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

Proof Generation. Therefore, the proof goal (Goal) for symbolic execution is proved in three phases:

Phase 1. Prove (Branch$_{j,l}$) and (Remainder$_j$) for each step j and branch $1 \leq l \leq l_j$.
Phase 2. Combine (Branch$_{j,l}$) and (Remainder$_j$) to obtain a proof of (Step$_j$).
Phase 3. Combine (Step$_j$) to obtain a proof of (Goal).

Remark 1 (Lemmas and Their Mechanized Proofs in Metamath)**.** We need many lemmas about the program execution relation "$\Rightarrow_{exec}$" when we generate proof certificates 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 Metamath formalization of the matching logic proof system [Chen et al. 2021a; K Team 2022b], as a part of the new contribution of the paper. These lemmas can be easily reused for future development.

In the following, we explain each proof generation step.

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

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

From the proof hint, we know that the corresponding substitution is $\theta_{j,l}$. Therefore, we instantiate the semantic rule using $\theta_{j,l}$ and obtain the following result

$$\Gamma^L \vdash lhs_{k_{j,l}}\theta_{j,l} \wedge q_{k_{j,l}}\theta_{j,l} \Rightarrow^1_{exec} rhs_{k_{j,l}}\theta_{j,l} \tag{6}$$

where we use $t\theta$ to denote the result of applying the substitution θ to t. Note that $q_{k_{j,l}}\theta_{j,l}$ is a predicate on the free variables of Equation (6) that holds on the left-hand side, by propositional reasoning, it also holds on the right-hand side. Therefore, we prove that:

$$\Gamma^L \vdash lhs_{k_{j,l}}\theta_{j,l} \wedge q_{k_{j,l}}\theta_{j,l} \Rightarrow^1_{exec} rhs_{k_{j,l}}\theta_{j,l} \wedge q_{k_{j,l}}\theta_{j,l} \tag{7}$$

To proceed, we need the following lemma:

Lemma 4.1 ($\Rightarrow^1_{exec}$ 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 4.1 allows us to strengthen the left-hand side and/or weaken the right-hand side of an execution relation. Using Lemma 4.1, and by comparing our proof goal ($\text{Branch}_{j,l}$) with Equation (7), we only need to prove the following two implications between constrained terms, which we call *subsumptions*:

$$\underbrace{\Gamma^L \vdash \left(t^{\text{hint}}_j \wedge p^{\text{hint}}_{j,l}\right) \rightarrow \left(lhs_{k_{j,l}}\theta_{k_{j,l}} \wedge q_{k_{j,l}}\theta_{k_{j,l}}\right)}_{\text{left-hand side strengthening}} \quad \underbrace{\Gamma^L \vdash \left(rhs_{k_{j,l}}\theta_{k_{j,l}} \wedge q_{k_{j,l}}\theta_{k_{j,l}}\right) \rightarrow \left(t^{\text{hint}}_{j,l} \wedge p^{\text{hint}}_{j,l}\right)}_{\text{right-hand side weakening}}$$

These subsumption proofs are common in our proof generation procedure (e.g. (Remainder_j) is also a subsumption). We elaborate on subsumption proofs in Section 4.3.

Phase 2: Proving (Step_j). We combine the proofs for each branch and the remainder as follows:

$$\Gamma^L \vdash t^{\text{hint}}_j \wedge p^{\text{hint}}_{j,1} \Rightarrow^1_{exec} t^{\text{hint}}_{j,1} \wedge p^{\text{hint}}_{j,1} \qquad (\text{Branch}_{j,1})$$

$$\vdots$$

$$\Gamma^L \vdash t^{\text{hint}}_j \wedge p^{\text{hint}}_{j,l_j} \Rightarrow^1_{exec} t^{\text{hint}}_{j,l_j} \wedge p^{\text{hint}}_{j,l_j} \qquad (\text{Branch}_{j,l_j})$$

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

Note that our proof goal (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 4.2 ($\Rightarrow_{exec}$ 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:

$$\Gamma^L \vdash p_j^{\text{hint}} \rightarrow p_{j,1}^{\text{hint}} \vee \cdots \vee p_{j,l_j}^{\text{hint}} \vee p_j^{\text{rem}} \tag{8}$$

The above implication includes only logical constraints and no configuration terms, and thus involves only *domain reasoning*. Therefore, we translate it into an equivalent FOL formula and delegate it to SMT solvers, such as Z3 [De Moura and Bjørner 2008].

From Equation (8), we can prove that the left-hand side of (Step_j), $t_j^{\text{hint}} \wedge p_j^{\text{hint}}$, can be broken down into $l_j + 1$ branches by propositional reasoning:

$$\Gamma^L \vdash \left(t_j^{\text{hint}} \wedge p_j^{\text{hint}}\right) \rightarrow \left(t_j^{\text{hint}} \wedge p_{j,1}^{\text{hint}}\right) \vee \ldots \vee \left(t_j^{\text{hint}} \wedge p_{j,l_j}^{\text{hint}}\right) \vee \left(t_j^{\text{hint}} \wedge p_j^{\text{rem}}\right) \tag{9}$$

Note that the right-hand side of Equation (9) is exactly the disjunction of all the left-hand sides of ($\text{Branch}_{j,l}$) and (Remainder_j). Therefore, to prove the proof goal (Step_j), we use the following lemma, which allows us to combine the executions in different branches into one (we will also need a consequence rule for $\Rightarrow_{exec}$ like Lemma 4.1, which is derivable from Lemmas 4.1 and 4.2):

Lemma 4.3 ($\Rightarrow_{exec}$ 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 generate the final proof certificate for symbolic execution. At a high level, the proof 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:

$$\Gamma^L \vdash (t \wedge p) \Rightarrow_{exec} (t \wedge p) \tag{10}$$

Then, we repeatedly apply the following steps to *symbolically execute* the right-hand side of Equation (10), until it becomes the same as the right-hand side of (Goal):

(1) Suppose we have obtained a proof certificate for

$$\Gamma^L \vdash (t \wedge p) \Rightarrow_{exec} \left(t_1^{\text{im}} \wedge p_1^{\text{im}}\right) \vee \cdots \vee \left(t_m^{\text{im}} \wedge p_m^{\text{im}}\right) \tag{11}$$

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

(2) Look for a (Step_j) claim of the form

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

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., the first intermediate constrained term $t_1^{\text{im}} \wedge p_1^{\text{im}}$ can be rewritten/executed using (Step_j).

(3) Symbolically execute $t_1^{\text{im}} \wedge p_1^{\text{im}}$ in Equation (11) for one step by applying (Step_j), and obtain the following proof:

$$\Gamma^L \vdash (t \wedge p) \Rightarrow_{exec} \underbrace{\left(t_{j,1}^{\text{hint}} \wedge p_{j,1}^{\text{hint}}\right) \vee \cdots \vee \left(t_{j,l_j}^{\text{hint}} \wedge p_{j,l_j}^{\text{hint}}\right) \vee \left(t_j^{\text{rem}} \wedge p_j^{\text{rem}}\right)}_{\text{right-hand side of } (\text{Step}_j)} \vee \underbrace{\left(t_2^{\text{im}} \wedge p_2^{\text{im}}\right) \vee \ldots \vee \left(t_m^{\text{im}} \wedge p_m^{\text{im}}\right)}_{\text{same as Equation (11)}}$$

Finally, after 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 generate a proof certificate for (Goal). Otherwise, the proof generation method fails, indicating potential mistakes made by $\mathbb{K}$'s symbolic execution engine.

4.3 Generating Proofs for Pattern Subsumption

It is common in generating proof certificates for symbolic execution that we need to generate the proof certificates for implications between constrained terms. We call such implications *subsumptions*. Formally, a subsumption has the form $\Gamma^L \vdash (t \wedge p) \rightarrow (t' \wedge p')$. We reduce it into the following two sub-goals that are sufficient for the subsumption to hold:

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

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

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

Then we call our proof generation method recursively on the above sub-goals. Note that the second type of sub-goals corresponds to the *unification* between t and t'.

Our method here for pattern subsumption is incomplete but covers most simplifications done by $\mathbb{K}$. Generally speaking, it is undecidable to prove such subsumptions as it requires to prove first-order theorems in an initial algebra of an equational/algebraic specification. However, there exist techniques that are shown to be effective in automating *inductive theorem proving*, such as Maude ITP [Hendrix 2008], which can be integrated by our work in the future.

4.4 Generating Proofs for Coinduction

Recall that the verification algorithm (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 4.2 and 4.3 can cover symbolic execution already, the missing part is line 12 in Algorithm 1, where we apply not the semantic rules but the claims in R to perform symbolic execution, which forms a circular argument. Our purpose is to generate proof certificates that justify the soundness of such circular reasoning, by showing that the algorithm is performing a coinduction on the (potentially infinite) execution trace.

We start with the simplest case when R has only one claim $\varphi \Rightarrow_{reach} \psi$. We assume that we have already rewritten φ to some intermediate configuration φ' using at least one steps (so logically

speaking, the set of claims $R = \{\varphi \Rightarrow_{reach} \psi\}$ has been flushed to the reachability logic axiom set by (Transitivity) in Figure 2):

$$\Gamma^L \vdash \varphi \Rightarrow^{+}_{reach} \varphi' \tag{12}$$

Further, suppose that the proof hint indicates that we need to apply the original claim $\varphi \Rightarrow_{reach} \psi$ (as a coinduction hypothesis) to φ'. We generate a proof certificate for this single step

$$\Gamma^L \vdash \Box(\forall FV(\varphi, \psi).\ \varphi \Rightarrow_{reach} \psi) \rightarrow \varphi' \Rightarrow_{reach} \varphi'' \tag{13}$$

where $FV(\varphi, \psi)$ is the set of all free variables in φ and ψ. Intuitively, we instantiate all the free variables using the substitution specified by the proof hint, where φ'' is the result of applying the claim $\varphi \Rightarrow_{reach} \psi$ as a regular semantic rule on φ'. Recall that Equation (13) is the matching logic encoding of the reachability judgment $\{\varphi \Rightarrow_{reach} \psi\} \vdash^{reach}_{\emptyset} \varphi' \Rightarrow \varphi''$ (see Example 5 and the discussion followed).

Now, we apply (Transitivity) to Equations (12) and (13) and obtain the proof certificate for

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

which is the matching logic encoding of the reachability judgment $\vdash^{reach}_{\{\varphi \Rightarrow_{reach} \psi\}} \varphi \Rightarrow \varphi''$, where $\varphi \Rightarrow_{reach} \psi$ belongs to the circularity set. Then, we reuse the proof generation procedure in Section 4.2 to generate the proof certificate for the symbolic execution of φ'', except that now there is an additional premise $\circ\Box(\forall FV(\varphi, \psi).\ \varphi \Rightarrow_{reach} \psi)$ that encodes the semantics of circularity.

Finally, if the verification algorithm successfully terminates, we will obtain the proof certificate

$$\Gamma^L \vdash \circ\Box(\forall FV(\varphi, \psi).\ \varphi \Rightarrow_{reach} \psi) \rightarrow \varphi \Rightarrow_{reach} \psi$$

which by (Circularity), derives $\Gamma^L \vdash \varphi \Rightarrow_{reach} \psi$, as desired.

Generally speaking, Algorithm 1 allows to 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 is called *set circularity*, which is derivable in reachability logic (see [Roşu et al. 2012, Lemma 5])

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

Here, all the claims in R are simultaneously added to the circularity set, featuring a mutual coinduction among all the coinduction hypotheses. Our current implementation does not support (Set Circularity) in its full generality. We assume that the proof of each claim only invokes itself as the coinduction hypothesis. This is not a restriction in theory because using [Roşu et al. 2012, Lemma 5], any proof using (Set Circularity) can be mechanically translated to one using only (Circularity), which is fully supported by our implementation.

5 IMPLEMENTATION

We implemented the proof generation procedures in Section 4 in Python. Our implementation can be found in [Lin et al. 2022]. Here we provide some interesting details about the implementation and discuss its limitations.

Firstly, we implemented a higher-level tactic language for writing proofs about types/sorts, from which the lower-level Metamath proofs are constructed. Note that $\mathbb{K}$ operates in a sorted setting while matching logic is unsorted. Instead, sorts are defined axiomatically using theories. To bridge this gap and reduce human engineering effort, we developed and used the tactic language to automate the generation of all the sort-related proofs. For example, to specify that the free variables x and y in a pattern φ have sorts s_1 and s_2, respectively, we write $\vdash (x{:}s_1 \wedge y{:}s_2) \rightarrow \varphi$, where $x{:}s_1$ and $y{:}s_2$ are predicates, stating that x and y belong to the inhabitants of s_1 and s_2, respectively.

Now, suppose we have proved $\vdash x{:}s_1 \to \psi$ and $\vdash (y{:}s_2 \wedge x{:}s_1) \to (\psi \to \varphi)$ and we want to prove $\vdash (x{:}s_1 \wedge y{:}s_2) \to \varphi$ using the following propositional lemma:

$$\frac{\vdash \theta \to \varphi \quad \vdash \theta \to (\varphi \to \psi)}{\vdash \theta \to \psi}$$

The tactic language will automatically rearrange the sort premises by proving that $\vdash (x{:}s_1 \wedge y{:}s_2) \leftrightarrow y{:}s_2 \wedge x{:}s_1$. A lot of such simple but tedious sort-related proofs are handled by the tactic language.

Secondly, we developed a library of 196 lemmas about the rewriting and reachability relations such as Lemma 4.2 in Section 4. These lemmas were proved manually in Metamath in ~4,000 lines and have been added to the existing Metamath database of matching logic. Note that all these lemmas are checked by the Metamath verifiers so they do not belong to the trust base.

Thirdly, we implemented several optimizations for constructing proof certificates to improve performance. To avoid reproducing a (sub)-proof over and over again, we cache an incomplete work-in-progress proof when its size exceeds a certain threshold and add it as a lemma, which can be used in future proofs to reduce duplicates. To save runtime memory, we represent proof trees as directed acyclic graphs (DAGs) where the common subtrees are shared. When we apply an intermediate lemma or combine multiple DAGs, we use a greedy algorithm to merge the subtrees that have the same conclusion. Even with these optimizations, proofs are still huge (in the order of tens of megabytes), which is primarily due to the space-inefficient text-based encoding. To reduce the proof sizes further, we can compress the proofs using a generic compression tool such as `xz` [Tukaani Team 2021], which provides >95% reduction in size; see Section 6 for more details.

The $\mathbb{K}$ deductive verifier consists of a powerful symbolic execution tool that supports many complex features such as evaluation order, conditional rewriting, "otherwise" rules (which are catch-all rules if no other semantic rules can be applied), user-defined contexts, unification modulo axioms, etc. Our current prototype implementation supports proof generation for a significant subset of these features. For evaluation orders, $\mathbb{K}$ specifies them using strictness attributes (Section 3.2), which are reduced to a special case of conditional rewriting, which is supported by our tool. The "otherwise" rules are also reduced to conditional rewriting where the condition states that no other semantic rules are applicable, and thus are also supported by our tool. $\mathbb{K}$ also provides a more advanced (but also much less often used) way to define evaluation orders using explicit user-defined contexts, which is not supported by our tool yet. Finally, unification modulo maps (i.e., unification modulo associativity, commutativity, and units) is supported. Currently, the logical encoding of a $\mathbb{K}$ semantics is computed by a frontend tool called `kompile` (see Figure 5), which lacks a clear documentation of the axioms it generates. This makes developing the proof generation procedure harder because we need to manually find suitable classes of axioms in `kompile`'s output. Therefore, we expect supporting proof generation for large real-world $\mathbb{K}$ developments to be a long-term endeavor, which involves a formalization of `kompile` and requires a close collaboration with the $\mathbb{K}$ team (see Section 7.1 for more discussion on `kompile`).

6 EVALUATION

We evaluated our proof generation method using two benchmark sets. The first benchmark set 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 set is a selection of C verification examples from the SV-COMP competition [SV-COMP 2021]. We used a machine with Intel i7-12700K processors and 32 GB of RAM. The evaluation results are shown in Figure 4. In the following, we discuss the benchmark sets and the evaluation results in detail.

Task	Spec. LOC	Steps	Hint Size	Proof Size	Time (seconds)			
					$\mathbb{K}$ Verifier	Gen.	Check 1	Check 2
`sum.imp`	40	42	0.58 MB	37/1.6 MB	4.2	105	1.8	9.6
`sum.reg`	46	108	2.24 MB	111/3.6 MB	9.1	259	5.4	15.9
`sum.pcf`	18	22	0.29 MB	38/1.5 MB	2.9	119	2.4	12.2
`exp.imp`	27	31	0.5 MB	37/1.5 MB	3.7	108	2.0	10.5
`exp.reg`	27	43	0.96 MB	70/2.3 MB	4.7	177	3.1	13.3
`exp.pcf`	20	29	0.5 MB	65/2.3 MB	3.8	199	3.1	13.7
`collatz.imp`	25	55	1.14 MB	49/1.7 MB	4.8	138	2.6	12.4
`collatz.reg`	37	100	3.66 MB	209/4.7 MB	9.3	414	5.5	31.6
`collatz.pcf`	26	39	1.51 MB	110/2.2 MB	5.3	247	5.2	23.6
`product.imp`	44	42	0.62 MB	44/1.8 MB	3.9	124	2.4	11.0
`product.reg`	24	42	0.81 MB	65/2.3 MB	4.3	164	4.0	11.8
`product.pcf`	21	48	0.82 MB	80/2.8 MB	5.3	234	4.9	18.4
`gcd.imp`	51	93	1.9 MB	74/2.3 MB	22.9	237	2.7	17.8
`gcd.reg`	27	73	1.92 MB	124/3.3 MB	18.6	306	3.6	16.9
`gcd.pcf`	22	38	1.35 MB	150/3.2 MB	12.8	367	5.2	28.5
`ln/count-by-1`	44	25	0.24 MB	28/1.3 MB	2.7	81	1.6	8.0
`ln/count-by-2`	44	25	0.26 MB	28/1.3 MB	9.0	88	1.4	8.1
`ln/gauss-sum`	51	39	0.53 MB	38/1.6 MB	4.6	107	2.0	10.2
`ln/half`	62	65	1.3 MB	63/2.2 MB	13.1	173	3.0	11.8
`ln/nested-1`	92	84	1.88 MB	104/3.4 MB	7.5	231	5.9	20.1

Fig. 4. Performance of Our Proof Generation Prototype. From left to right, we list the verification tasks, **spec**ification **LOC**, number of symbolic execution **steps**, proof **hint size**, **proof** object **size** (uncompressed/-compressed), $\mathbb{K}$ verifier time (without proof generation), proof **gen**eration time, and proof **check**ing time (check 1 using `smetamath` [O'Rear and Carneiro 2019] and check 2 using our own implementation in Rust [Wang 2022]). Tasks with prefix `ln/` are from the `loop-new` benchmark of SV-COMP [SV-COMP 2021].

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 C-like syntax;
- REG: an assembly language for a register-based virtual machine;
- PCF, i.e., programming computable functions [Plotkin 1977]: a typed functional language with a fixed-point operator.

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 [Guy 2004] 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.

All benchmark programs and their formal specifications are implemented/specified in the three programming languages IMP, REG, and PCF. Figure 4 shows that our prototype can generate proof certificates for all these programs without additional effort. The detailed encoding of these verification tasks in $\mathbb{K}$ can be found in our repository [Lin et al. 2022]. Besides these verification

examples, we also considered the C programs from the `loop-new` benchmark set in the SV-COMP competition [SV-COMP 2021].

Even for simple arithmetic programs such as `SUM`, the symbolic execution process is complicated, as one can see from the proof object sizes in Figure 4. A lot of seemingly innocuous operations that are performed by the $\mathbb{K}$ deductive verifier, such as substitution and equational simplification, result in very long matching logic proof certificates, which encode proof steps down to the lowest possible level—the proof system (Figure 3).

Evaluation Results. We measured the performance of both proof generation and proof checking. For proof generation, we measured the generation time, the number of symbolic execution steps, the sizes of the proof hint and the final proof certificates. We also measured the sizes of compressed proof certificates using a generic compression tool `xz` [Tukaani Team 2021]; these compressed proofs can be decompressed and checked on-the-fly using an online Metamath verifier such as `mmverify` [Levien and Wheeler 2019]. The *key highlights* of our evaluation are:

(1) Proof checking using Metamath is very fast, even for very long proofs;.
(2) Proof generation takes more time, often in the order of minutes, depending on the number of symbolic execution steps that are conducted during verification but not much relevant to the size of the program being verified.
(3) Proof certificates are very large, but they are simply plain text files and their sizes can be greatly reduced using any mainstream compression tool. Compressed proofs can be decompressed on-the-fly for proof checking by using an online Metamath verifier, as a space-time trade-off.

We explain the experimental results in detail.

Proof Generation. At a high level, the proof generation time consists of (1) the time to generate the matching logic theory Γ^L from the $\mathbb{K}$ formal language semantics of L, and (2) the time to generate the proof certificates using the procedures described in Section 4. In our experiments, (1) only takes a few seconds and is linear to the number of semantic rules. Most time is spent on (2), which is linear to the number of symbolic execution steps conducted during verification and the sizes of the intermediate configurations. Generally speaking, deductive verifiers are slow, and it takes even more time for users to propose the right invariants. In our view, it is therefore acceptable to spend the extra time on generating rigorous and machine-checkable proof certificates for deductive verifiers and their verification runs, which help establish the correctness of the verification results on a smaller trust base.

Proof Checking. Due to the simplicity of Metamath and the 240-line formalization of matching logic, it is very fast to check proof certificates. 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 certificates, set up a proof checking environment (which is much simpler than setting up a verification environment), and check the proofs independently. We are optimistic about the scalability of our method on large $\mathbb{K}$ developments because proof checking scales well. The sizes of proof certificates are linear to the number of symbolic execution steps and the sizes of configurations. The complexity of proof checking is also linear to the sizes of proof certificates. We do not see a nonlinear factor or an exponential explosion in our proof generation method.

Proof Compression. Metamath has its own format to compress proofs (see [Megill and Wheeler 2019, Appendix B]). On top of that, proof certificates can be compressed as plain text files using any mainstream compression tool such as `xz` [Tukaani Team 2021], which leads to >95% reduction

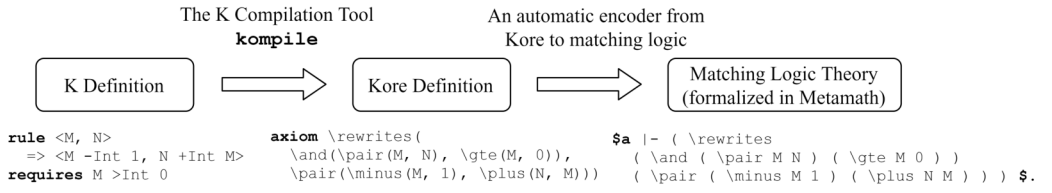

Fig. 5. Two-Phase Translation from $\mathbb{K}$ to Matching Logic

in the proof sizes, as shown in Figure 4, at the expense of spending more time in decompressing the proofs for proof checking and using an online proof checker, which can be slower than an offline one. It is left as future work to study such space-time trade-off in proof checking and find the right balance.

7 DISCUSSION

7.1 Reducing the Trust Base of $\mathbb{K}$

$\mathbb{K}$ is a complicated artifact under active development. Among its 550,000 lines of code base, roughly 40,000 lines are for the frontend, implemented in Java. There is also 160,000 lines of C++/Java code that focuses mainly on efficient concrete program execution. The most relevant code base is the 120,000-line Haskell backend that supports symbolic reasoning and formal verification. The language-agnostic deductive verifier is implemented in the Haskell backend of $\mathbb{K}$.

The $\mathbb{K}$ frontend provides an intuitive frontend syntax that allows to write formal semantics more easily. For example, the frontend syntax swallows the entire concrete syntax of the programming language being defined and allows language designers to use directly the concrete syntax in writing the semantic rules, without needing to write their abstract syntax trees. Also, the frontend syntax includes shortcuts and notations for writing program configurations. In a semantic rule, only the necessary part of a configuration needs to be explicitly mentioned, while the other part can be omitted and automatically inferred by $\mathbb{K}$. The frontend also implements type inference for the variables in semantic rules, so the users usually do not need to explicitly specify the variable types.

All the above frontend shortcuts and notations will be eliminated by the frontend of $\mathbb{K}$. The frontend tool `kompile` translates the formal language semantics into an intermediate formal language called Kore [K Team 2022a], which is used to specify matching logic patterns and axioms. `kompile` parses all the concrete syntax into abstract syntax trees, represented as patterns. It also infers the omitted parts of configurations in semantic rules and the types of all the variables. In the end, `kompile` produces one Kore definition— as one source file `definition.kore`—that includes the entire matching logic encoding of the formal language semantics. The compiled Kore file is then passed to $\mathbb{K}$'s backends to generate the corresponding language tools.

Therefore, Kore behaves as the intermediate interface between the frontend and the backends. It is also the boundary between the informal and formal worlds. Since Kore is a formal specification language for writing matching logic theories, the formal semantics of a Kore definition *is* the matching logic theory that it defines. However, the frontend syntax of $\mathbb{K}$ (as shown in Figure 1) does not (yet) have a formal semantics. Its meaning is completely determined by `kompile`, which lacks a formal specification.

In this paper, we are interested in certifying *backend* correctness. More precisely, we are certifying the language-agnostic deductive verifier, implemented by the Haskell backend. Previously, the correctness of formal verification in $\mathbb{K}$ depends on the 120,000-line Haskell backend and its internal verification algorithm (Algorithm 1) as well as optimized, complex algorithms for symbolic execution

and pattern matching/subsumption. By generating proof certificates for these algorithms, we eliminate them from the trust base.

We should also clarity that the entire trust base for *end-to-end* verification in $\mathbb{K}$ is still large and should be further reduced in the future. Firstly, the `kompile` tool belongs to the trust base. Secondly, the automatic encoder (developed in [Chen et al. 2021a]) that translates Kore into Metamath belongs to the trust base (Figure 5), although the translation is very simple; it only parses the Kore definition and prints it in the Metamath format. Thirdly, the formalization of matching logic in Metamath belongs to the trust base, which is very small (240 lines). However, all the *backend* algorithms are no longer in the trust base. They are certified by matching logic proofs and the proof checker.

7.2 Future Directions

We identify some main future directions of the current work.

Firstly, as discussed in Section 7.1, the frontend tool `kompile` needs to be trusted. It is not satisfying, because the frontend consists of roughly 40,000 lines of Java, while many tasks that it performs, such as configuration inference and completion, can also be formalized as matching logic proofs, the same way how program execution and deductive verification are matching logic proofs. In the long run, we see no reason to not formalize the *entire* $\mathbb{K}$ frontend, even including the parser. Indeed, the concrete syntax given by a context-free grammar can be regarded as the initial algebra of an equational/algebraic specification [Goguen et al. 1977]. A parser can then be specified as a function from the domain of strings (sequences of characters) to that initial algebra. Since initial algebra semantics can be defined in matching logic [Chen et al. 2020], the parsing function can be inductively axiomatized and certified by matching logic proofs.

The second future direction is to incorporate proofs for SMT solvers. Currently, our implementation trusts SMT solvers and does not generate proof objects for them. $\mathbb{K}$ uses SMT solvers for domain reasoning, such as $\Gamma^L \vdash \varphi \longrightarrow \psi$, where φ and ψ 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, thus resulting in a gap in our proof certificates that needs to be addressed separately in the future, following existing research such as [Barrett et al. 2015; Stump et al. 2013].

The third future direction is to address the current incompleteness of the proof generation procedure (i.e. failure to produce a proof even when the verifier succeeds). Currently, we can identify two sources of incompleteness:

- The subsumption proof generation (Section 4.3) may not match the actual simplification procedure of the $\mathbb{K}$ verifier, thus resulting in subsumptions that are correctly done by $\mathbb{K}$ but cannot be proved by our proof generation tool.
- Our proof generation procedure does not support the (Set Circularity) rule as discussed in Section 4.4, while the $\mathbb{K}$ verifier does use (Set Circularity) in general.

These sources of incompleteness arise from the inconsistency between our proof generation procedure and the actual implementation of the $\mathbb{K}$ verifier. Therefore, a long-term collaboration with the $\mathbb{K}$ team is required to improve the completeness of our proof generation tool.

Finally, as discussed in Section 4.1, we plan to extend our proof generation method to support proof generation for all-path reachability reasoning [Ştefănescu et al. 2014, 2016]. In the current work, we only consider one-path reachability logic, which captures the partial correctness of one execution trace. For nondeterministic and concurrent programs, we need all-path reachability logic to prove the correctness of all execution traces. All-path reachability logic is proposed for precisely that purpose. An all-path reachability claim $\varphi \Rightarrow^{\forall}_{reach} \psi$ holds iff for every maximal and finite execution traces starting from φ, ψ is reachable. The proof system of all-path reachability logic has identical proof rules as one-path reachability logic in Figure 2 (replacing $\Rightarrow$ with $\Rightarrow^{\forall}_{reach}$), except

one additional axiom called (Step)

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

where $A = \{lhs_1 \Rightarrow rhs_1, \ldots, lhs_K \Rightarrow rhs_K\}$ is the set of all the semantic rules, which are one-path rules in nature. The (Step) axiom derives all-path claims from these semantic rules, where ψ_k is the result of executing φ for one step, using the k-th semantic rule $lhs_k \Rightarrow rhs_k$ for $1 \leq k \leq K$. Thus, the (Step) axiom states that the only way to make an execution step is to use one of the semantic rules in A. Since the current $\mathbb{K}$ pipeline that translates $\mathbb{K}$ into matching logic (Figure 5) is incomplete and the resulting theory Γ^L does not have the (Step) axiom, proof generation for all-path reachability claims is left as future work.

8 CONCLUSION

In this paper, we proposed a method that generates proof certificates for the language-agnostic one-path deductive verifier in the $\mathbb{K}$ formal semantics framework. Each successful run of the verifier is certified by a formal proof in matching logic, on a case-by-case basis. The proof certificates consist of the entire formal semantics of the programming language as matching logic axioms and the program property being verified, as well as the detailed proof steps that derive the property from the formal language semantics. Our proof certificates are encoded in Metamath and can be automatically checked by any Metamath verifiers. We finished a prototype implementation of our proof generation method and experimented with it on verification examples across three different programming languages, which demonstrated that our method supports *language-agnostic* verification. The experiment showed promising performance in both proof generation and proof checking. With the proposed work, we reduced the trust base of $\mathbb{K}$. What was previously in the trust base—the internal algorithms for verification, symbolic execution, pattern matching, etc. in the backend of $\mathbb{K}$, comprising 120,000 lines of Haskell—are now certified by proof certificates.

DATA AVAILABILITY STATEMENT

We have prepared a publicly available Docker image [Lin et al. 2023] for reproducing our evaluations in Section 6.

ACKNOWLEDGMENTS

The work presented in this paper was supported in part by an IOHK grant and an Ethereum Foundation gift. This material is based upon work supported by the United States Air Force and DARPA under Contract No. FA8750-18-C-0092.

REFERENCES

Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2006. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 364–387. https://doi.org/10.1007/11804192_17

Clark Barrett, Leonardo De Moura, and Pascal Fontaine. 2015. Proofs in satisfiability modulo theories. Available at http://leodemoura.github.io/files/SMTProofs.pdf. *All about proofs, Proofs for all* 55, 1 (2015), 23–44.

Sandrine Blazy and Xavier Leroy. 2009. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning* 43, 3 (2009), 263–288. https://doi.org/10.1007/s10817-009-9148-3

Denis Bogdănaş and Grigore Roşu. 2015. K-Java: A complete semantics of Java. In *Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL'15)*. ACM, Mumbai, India, 445–456. https://doi.org/10.1145/2775051.2676982

Mario Carneiro. 2020. Metamath Zero: Designing a Theorem Prover Prover. In *Intelligent Computer Mathematics*, Christoph Benzmüller and Bruce Miller (Eds.). Springer International Publishing, Cham, 71–88. https://doi.org/10.1007/978-3-030-53518-6_5

Xiaohong Chen, Zhengyao Lin, Minh-Thai Trinh, and Grigore Roşu. 2021a. Towards a Trustworthy Semantics-Based Language Framework via Proof Generation. In *Proceedings of the 33rd International Conference on Computer-Aided Verification*. ACM, Virtual, 22 pages. https://doi.org/10.1007/978-3-030-81688-9_23

Xiaohong Chen, Dorel Lucanu, and Grigore Roşu. 2020. *Initial algebra semantics in matching logic.* Technical Report. University of Illinois at Urbana-Champaign. http://hdl.handle.net/2142/107781

Xiaohong Chen, Dorel Lucanu, and Grigore Roşu. 2021b. Matching logic explained. *Journal of Logical and Algebraic Methods in Programming* 120 (2021), 1–36. https://doi.org/10.1016/j.jlamp.2021.100638

Xiaohong Chen and Grigore Roşu. 2019a. Matching μ-logic. In *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'19)*. IEEE, Vancouver, Canada, 1–13. https://doi.org/10.1109/LICS.2019.8785675

Xiaohong Chen and Grigore Roşu. 2019b. *Matching μ-logic.* Technical Report. University of Illinois at Urbana-Champaign. http://hdl.handle.net/2142/102281

Xiaohong Chen and Grigore Roşu. 2020. A general approach to define binders using matching logic. In *Proceedings of the 25th ACM SIGPLAN International Conference on Functional Programming (ICFP'20)*. ACM, New Jersey, USA, 1–32. https://doi.org/10.1145/3408970

Coq Team. 2021a. Coq GitHub Repository. https://github.com/coq/coq.

Coq Team. 2021b. *The Coq proof assistant.* Inria. http://coq.inria.fr

Andrei Ştefănescu, Ştefan Ciobâcă, Radu Mereuţă, Brandon M. Moore, Traian Florin Şerbănuţă, and Grigore Roşu. 2014. 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)*, Vol. 8560. Springer, Vienna, Austria, 425–440. https://doi.org/10.1007/978-3-319-08918-8_29

Andrei Ştefănescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore Roşu. 2016. 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)*. ACM, Amsterdam, Netherlands, 74–91. https://doi.org/10.1145/3022671.2984027

Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S. Adve, and Grigore Roşu. 2019. 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)*. ACM, Phoenix, Arizona, USA, 1133–1148. https://doi.org/10.1145/3314221.3314601

Leonardo De Moura and Nikolaj Bjørner. 2008. 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)*, Vol. 4963. Springer, Budapest, Hungary, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Chucky Ellison and Grigore Rosu. 2012. An executable formal semantics of C with applications. *ACM SIGPLAN Notices* 47, 1 (2012), 533–544. https://doi.org/10.1145/2103621.2103719

Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 — Where Programs Meet Provers. In *Programming Languages and Systems*, Matthias Felleisen and Philippa Gardner (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 125–128. https://doi.org/10.1007/978-3-642-37036-6_8

Quentin Garchery. 2021. A Framework for Proof-carrying Logical Transformations. *Electronic Proceedings in Theoretical Computer Science* 336 (July 2021), 5–23. https://doi.org/10.4204/eptcs.336.2

Joseph Goguen, James Thatcher, Eric Wagner, and Jesse Wright. 1977. Initial algebra semantics and continuous algebras. *Journal of the ACM* 24, 1 (1977), 68–95. https://doi.org/10.1145/321992.321997

Dwight Guth. 2013. *A formal semantics of Python 3.3.* Technical Report. University of Illinois at Urbana-Champaign. http://hdl.handle.net/2142/45275

Dwight Guth, Chris Hathhorn, Manasvi Saxena, and Grigore Roşu. 2016. RV-Match: Practical semantics-based program analysis. In *Proceedings of the 28th International Conference on Computer Aided Verification (CAV'16)*, Vol. 9779. Springer, Toronto, Ontario, Canada, 447–453. https://doi.org/10.1007/978-3-319-41528-4_24

Richard Guy. 2004. *Unsolved problems in number theory.* Vol. 1. Springer Science & Business Media, Berlin, Heidelberg. https://doi.org/10.1007/978-0-387-26677-0

Robert Harper, David MacQueen, and Robin Milner. 1986. *Standard ML.* Department of Computer Science, University of Edinburgh, Edinburgh, UK. http://www.lfcs.inf.ed.ac.uk/reports/86/ECS-LFCS-86-2/

Joseph D Hendrix. 2008. *Decision procedures for equationally based reasoning.* Ph. D. Dissertation. University of Illinois at Urbana-Champaign. http://hdl.handle.net/2142/11487

Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, Brandon Moore, Yi Zhang, Daejun Park, Andrei Ştefănescu, and Grigore Roşu. 2018. KEVM: A complete semantics of the Ethereum virtual machine. In *Proceedings of the 2018 IEEE Computer Security Foundations Symposium (CSF'18)*. IEEE, Oxford, UK, 204–217. https://doi.org/10.1109/CSF.2018.00022

C. A. R. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580. https://doi.org/10.1145/363235.363259

Isabelle Team. 2021. Isabelle. https://isabelle.in.tum.de/.

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods*, Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 41–55. https://doi.org/10.1007/978-3-642-20398-5_4

K Team. 2022a. K framework Haskell backend. https://github.com/kframework/kore.

K Team. 2022b. Matching logic proof checker. GitHub page https://github.com/kframework/proof-generation. See https://github.com/kframework/proof-generation/blob/main/theory/matching-logic-240-loc.mm for the 240-line formalization of matching logic in Metamath..

Theodoros Kasampalis, Daejun Park, Zhengyao Lin, Vikram S Adve, and Grigore Roşu. 2021. 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*. ACM New York, NY, USA, Virtual, 1004–1019. https://doi.org/10.1145/3445814.3446751

Dexter Kozen. 1983. Results on the propositional μ-calculus. *Theoretical Computer Science* 27, 3 (1983), 333–354. https://doi.org/10.1016/0304-3975(82)90125-6

Ramana Kumar, Magnus O Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. *ACM SIGPLAN Notices* 49, 1 (2014), 179–191. https://doi.org/10.1145/2578855.2535841

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

Raph Levien and David A. Wheeler. 2019. Metamath Verifier in Python. https://github.com/david-a-wheeler/mmverify.py.

Liyi Li and Elsa Gunter. 2020. K-LLVM: A Relatively Complete Semantics of LLVM IR. In *Proceedings of the 34th European Conference on Object-Oriented Programming (ECOOP 2020)*. ACM New York, NY, USA, Virtual, 1–29. https://doi.org/10.4230/LIPIcs.ECOOP.2020.7

Zhengyao Lin, Xiaohong Chen, Minh-Thai Trinh, John Wang, and Grigore Roşu. 2022. K Proof Generation Tool Repository. https://github.com/kframework/proof-generation.

Zhengyao Lin, Xiaohong Chen, Minh-Thai Trinh, John Wang, and Grigore Roşu. 2023. Generating Proof Certificates for a Language-Agnostic Deductive Program Verifier. https://doi.org/10.5281/zenodo.7503088

Qingzhou Luo, Yi Zhang, Choonghwan Lee, Dongyun Jin, Patrick O'Neil Meredith, Traian Florin Şerbănuţă, and Grigore Roşu. 2014. RV-Monitor: Efficient parametric runtime verification with simultaneous properties. In *Proceedings of the 5th International Conference on Runtime Verification (RV'14)*. Springer International Publishing, Toronto, Canada, 285–300. https://doi.org/10.1007/978-3-319-11164-3_24

Norman Megill and David A. Wheeler. 2019. Metamath: a computer language for mathematical proofs. Available at http://us.metamath.org/downloads/metamath.pdf.

George C Necula and Peter Lee. 2000. Proof generation in the Touchstone theorem prover. In *Proceedings of the 17th International Conference on Automated Deduction*. Springer, Springer-VerlagBerlin, Heidelberg, Pittsburgh, Pennsylvania, USA, 25–44. https://doi.org/10.1007/10721959_3

Stefan O'Rear and Mario Carneiro. 2019. Metamath Verifier in Rust. https://github.com/sorear/smetamath-rs.

Daejun Park, Andrei Ştefănescu, and Grigore Roşu. 2015. KJS: A complete formal semantics of JavaScript. In *Proceedings of the 36th annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*. ACM, Portland, OR, 346–356. https://doi.org/10.1145/2737924.2737991

Gaurav Parthasarathy, Peter Müller, and Alexander J. Summers. 2021. Formally Validating a Practical Verification Condition Generator. In *Computer Aided Verification*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer International Publishing, Cham, 704–727. https://doi.org/10.1007/978-3-030-81688-9_33

Gordon D. Plotkin. 1977. LCF considered as a programming language. *Theoretical computer science* 5, 3 (1977), 223–255. https://doi.org/10.1016/0304-3975(77)90044-5

A. Pnueli, M. Siegel, and E. Singerman. 1998. Translation validation. In *Tools and Algorithms for the Construction and Analysis of Systems*, Bernhard Steffen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 151–166. https://doi.org/10.1007/BFb0054170

Grigore Roşu. 2017. Matching logic. *Logical Methods in Computer Science* 13, 4 (2017), 1–61. https://doi.org/10.23638/LMCS-13(4:28)2017

Grigore Roşu, Andrei Ştefănescu, Ştefan Ciobâcă, and Brandon M. Moore. 2013. One-path reachability logic. In *Proceedings of the 28th Symposium on Logic in Computer Science (LICS'13)*. IEEE, New Orleans, USA, 358–367. https://doi.org/10.1109/LICS.2013.42

Grigore Roşu, Andrei Ştefănescu, Ştefan Ciobâcă, and Brandon M. Moore. 2012. *Reachability Logic*. Technical Report. University of Illinois at Urbana-Champaign. http://hdl.handle.net/2142/32952

Grigore Roşu and Wolfram Schulte. 2009. *Matching logic—extended report*. Technical Report. University of Illinois at Urbana-Champaign. https://fsl.cs.illinois.edu/publications/rosu-schulte-2009-tr.pdf

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

Konrad Slind and Michael Norrish. 2008. A brief overview of HOL4. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, Springer-Verlag Berlin Heidelberg, Montreal, Canada, 28–32. https://doi.org/10.1007/978-3-540-71067-7_6

Aaron Stump, Duckki Oe, Andrew Reynolds, Liana Hadarean, and Cesare Tinelli. 2013. SMT proof checking using a logical framework. *Formal Methods in System Design* 42, 1 (2013), 91–118. https://doi.org/10.1007/s10703-012-0163-3

SV-COMP. 2021. Benchmark for SV-COMP. https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks.

Alfred Tarski. 1955. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.* 5, 2 (1955), 285–309.

Tukaani Team. 2021. XZ Utils. https://tukaani.org/xz/.

John Wang. 2022. Metamath proof checker in Rust. GitHub page https://github.com/kframework/rust-metamath.

Stefan Wils and Bart Jacobs. 2021. Certifying C program correctness with respect to CompCert with VeriFast. https://doi.org/10.48550/ARXIV.2110.11034

Received 2022-10-28; accepted 2023-02-25