Technical Report:
Trustworthy Program Verification via Proof Generation

Zhengyao Lin
zl38@illinois.edu

Xiaohong Chen
xc3@illinois.edu

Minh-Thai Trinh
trinhmt@illinois.edu

John Wang
jzw2@illinois.edu

Grigore Roşu
grosu@illinois.edu

University of Illinois at Urbana-Champaign

July 18, 2021

Abstract

In an ideal language framework, language designers only need to define the formal semantics of their languages. Deductive program verifiers and other language tools are automatically generated by the framework. In this paper, we propose a novel approach to establishing the correctness of these autogener-erated verifiers via proof generation. Our approach is based on the $K$ language framework and its logical foundation, matching logic. Given a formal language semantics in $K$, we translate it into a corresponding matching logic theory. Then, we encode formal verification tasks as reachability formulas in matching logic. The correctness of one verification task is then established, on a case-by-case basis, by automatically generating a rigorous, machine-checkable mathematical proof of the associated reachability formula. Experiments with our proof generation prototype on various verification tasks in different programming languages show promising performance and attest to the feasibility of the proposed approach.

1 Introduction

Formal deductive verification aims at proving the correctness of algorithms and programs with respect to their formal specifications, using formal methods of mathematics. Traditional deductive verification is based on an axiomatic semantics of the programming language, such as Hoare logic [26]. More recent research shows that it is possible to automatically derive a sound and relatively complete deductive program verifier directly from the operational semantics of the programming language, making formal verification frameworks language-independent [13].
Such a language-independent approach towards formal verification fuels the vision of an ideal language framework. As shown in Figure 1, an ideal language framework is one where language designers only need to define the formal syntax and semantics of their language. All language tools of that language, including execution and formal analysis tools, are automatically generated by the framework.

As a continuing effort to realize the above vision, the \( \mathbb{K} \) framework (https://kframework.org) has shown increasing interest from the programming language community. It provides an intuitive meta-language for language designers to define the formal semantics of their programming languages. From such a formal semantics, \( \mathbb{K} \) automatically generates useful language tools, including parsers, interpreters, deductive verifiers [14], program equivalence checkers [28], among others. In practice, \( \mathbb{K} \) has been used to define the complete executable formal semantics of C [17], Java [4], JavaScript [35], Python [19], Ethereum virtual machine (EVM) [25], and x86-64 [15], from which their implementations and formal analysis tools are automatically generated. Some commercial products [20, 33] are powered by these autogenerated implementations and tools.

A major research question is: Should we trust these language tools generated by \( \mathbb{K} \)? Indeed, given the complexity of \( \mathbb{K} \)—a complex artifact with more than 500,000 lines of code written in 4 programming languages—we cannot take the correctness of \( \mathbb{K} \) for granted. Instead, its correctness must be justified by rigorous and machine-checkable mathematical proofs.

To address the above research question, we propose in [5] a proof generation framework for \( \mathbb{K} \) that aims at generating rigorous and machine-checkable mathematical proofs that justify the correctness of \( \mathbb{K} \)'s language tools. As a first step, we consider (concrete) program execution. Given the \( \mathbb{K} \) formal semantics of a programming language \( L \), we turn it into a logical theory \( \Gamma^L \) within matching logic—the logical foundation of \( \mathbb{K} \). Then, we generate a proof object for

\[
\Gamma^L \vdash \varphi_{init} \Rightarrow_{\text{exec}} \varphi_{final}
\]  

(1)

for a concrete execution trace from the initial state \( \varphi_{init} \) to the final state \( \varphi_{final} \). Here, “\( \Rightarrow_{\text{exec}} \)” means rewriting, i.e., program execution. The generated proof objects are encoded in and can be directly checked by Metamath—a small and fast third-party proof framework [36].

Figure 1: An ideal language framework vision; language tools are autogenerated, with machine-checkable mathematical proofs as correctness certificates (green gears represent accomplished items)
Our contribution in this paper is proof generation for K’s formal verification tool. We extend our previous work from concrete execution to symbolic execution, and further, to formal verification. Given a programming language L and a verification task, we generate a proof object for

\[ \Gamma^L \vdash \varphi_{\text{pre}} \Rightarrow_{\text{reach}} \varphi_{\text{post}} \]  

(2)

where \( \varphi_{\text{pre}} \) and \( \varphi_{\text{post}} \) are symbolic formulas for the pre- and post-conditions of the verification task, respectively. Here, “\( \Rightarrow_{\text{reach}} \)” is called reachability, which captures the notion of partial correctness. That is, divergent, infinite execution traces automatically satisfy reachability “\( \Rightarrow_{\text{reach}} \)”, and only convergent, finite execution traces are required to satisfy the post condition \( \varphi_{\text{post}} \). Proof obligations \( 1 \) and \( 2 \) are different because \( 2 \) allows divergent execution traces, which requires us to reason about repetitive behaviors of programs.

By encoding formal verification tasks as machine-checkable matching logic proof objects, we reduce the trust base of K’s formal verification tool dramatically, from the entire K code base of over 500,000 lines in multiple languages to much smaller components, consisting of a 240-line formalization of matching logic in Metamath (see Section 4.2) and the Metamath proof framework itself. This way, we obtain program verifiers with a small trust base.

It is worth mentioning that our approach is language-independent. The algorithm that generates the proof objects for program verification, such as Equation (2), is not specific to the language semantics \( \Gamma^L \), nor to the verification tasks given by \( \varphi_{\text{pre}} \) and \( \varphi_{\text{post}} \). Instead, our proof generation method is parametric in the formal languages semantics. This way, we obtain trustworthy program verifiers for all programming languages.

We implemented our proof generation method as a prototype and experimented with it on several verification examples across different programming languages. The results of our experiments show promising performance in both proof generation and proof checking. For example, we used K to verify a simple program \texttt{sum} that calculates the total sum from 1 to a symbolic input \( n \). The corresponding proof object is 128 MB large. It takes 2.0 seconds to proof-check the proof object on a regular laptop (see Section 8).

The rest of the paper is organized as follows. In Section 2, we give an overview of our approach. Then we introduce K and matching logic in Sections 3 and 4, respectively. We discuss reachability formulas in Section 5. Our main technical contribution—proof generation for symbolic execution and deductive verification—is discussed in Sections 6 and 7. We show the results of our experiments in Section 8 and discuss some limitations of our current prototype implementation in Section 9. Finally, we conclude the paper with related work in Section 10.

2 Overview of Our Approach

In this section, we discuss the interesting technical aspects of our approach. Recall that for every verification task conducted by K, we generate a proof for \( \Gamma^L \vdash \varphi_{\text{pre}} \Rightarrow_{\text{reach}} \varphi_{\text{post}} \) in matching logic. Our proof generation method consists of the following four components:

1. Matching logic and its proof system (i.e., “\( \vdash \)”), serving as the logical foundation of K;
2. Encoding of verification tasks as matching logic formulas, called reachability “\( \Rightarrow_{\text{reach}} \)”;
3. Proof generation for symbolic execution and circular proofs (for repetitive behaviors);
4. A third-party trustworthy proof framework called Metamath [36].

We explain these components in the following.

2.1 Matching Logic: Logical Foundation of K

Matching logic [40, 8, 6] is a simple mathematical logic that forms the logical foundation of K. By “logical foundation”, we mean the following:
1. The K formal semantics of a programming language L yields a matching logic theory Γ_L, which, roughly speaking, consists of a set of symbols that represent the syntax of L, and a set of axioms that specify the formal operational semantics of L.

2. Language tasks conducted by K can be formally specified by matching logic formulas. For example, deductive verification is specified (in our approach) by the following reachability formula:

\[ \varphi_{pre} \Rightarrow_{reach} \varphi_{post} \]  

(3)

where \( \varphi_{pre} \) and \( \varphi_{post} \) denote the (symbolic) pre- and post-conditions, and \( \Rightarrow_{reach} \) means partial functional correctness, which is intended in deductive verification (see Section 5).

3. Matching logic has a proof system that defines the provability relation \( \vdash \) between theories and formulas. For example, the correctness of the above verification task is justified by the following proof:

\[ \Gamma_L \vdash \varphi_{pre} \Rightarrow_{reach} \varphi_{post} \]  

(4)

Therefore, matching logic is the logical foundation of K. The correctness of the language tools generated by K is reduced to the existence of matching logic proofs, such as Equation (4). Such formal proofs are encoded as proof objects, which are automatically generated by our proof generation method and ready to be proof-checked by a third-party proof framework.

2.2 Encoding Deductive Verification as Reachability

Reachability logic, first proposed in [41] and further developed in [13], is a formal system for deductive verification, based on the operational semantics of programming languages (as opposed to axiomatic semantics, such as Hoare logic). It comes with a sound, relatively complete and language-independent proof system, which is implemented in K’s formal verification tool [14]. Recently, [8] shows that reachability logic can be fully defined in matching logic. Therefore, matching logic is powerful enough to serve as the logical foundation of K’s formal verification tool.

The intuitive meaning of the reachability formula \( \varphi_{pre} \Rightarrow_{reach} \varphi_{post} \) is as follows. For any initial state \( \gamma_0 \)—called a configuration in our approach—that satisfies \( \varphi_{pre} \), there exists an execution trace \( \tau = \gamma_0 \gamma_1 \gamma_2 \ldots \) such that if \( \tau \) is convergent/finite, then there exists \( n \geq 0 \) such that \( \gamma_n \) satisfies \( \varphi_{post} \). That is, \( \varphi_{post} \) should eventually hold on finite execution traces. It is known as partial correctness.

Although we write \( \varphi_{pre} \) and \( \varphi_{post} \), which are reminiscent of the pre- and post-conditions of a Hoare triple [26], it is worth mentioning that reachability is more flexible than Hoare triples because it does not enforce the execution traces to be complete (i.e., starting at an initial configuration and ending, if terminating, at a final configuration). Instead, it allows trace fragments that start and end at any intermediate configurations, and thus is more convenient to use in practice. Hoare triples can be regarded as a special case where \( \varphi_{pre} \) and \( \varphi_{post} \) can only be satisfied by the initial and final configurations, respectively.

One major contribution of this paper is that we implement the theoretical results in [8]. Specifically, a total of 191 matching logic lemmas have been encoded, and their detailed, machine-checkable proofs (of nearly 4,000 lines) have been completely worked out, forming a comprehensive database of lemmas for program verification (the source can be found in [2]). Our proof generation method (discussed in Section 2.3) is built upon these lemmas, some of which are presented later in this paper (Lemmas 4, 5 and 7 to 9).

The other major contribution is that we encode reachability reasoning into machine-checkable matching logic proof objects. At a high level, a reachability proof of \( \Gamma_L \vdash \varphi_{pre} \Rightarrow_{reach} \varphi_{post} \) is carried out as follows:

1. If \( \Gamma_L \vdash \varphi_{pre} \to \varphi_{post} \) then conclude the proof successfully (note that \( \to \) denotes propositional implication; that is, all configurations satisfying \( \varphi_{pre} \) already satisfy \( \varphi_{post} \)).

\(^1\)The completeness is relative to the domain reasoning of configurations. That is, given an oracle that can decide the validity of any formulas about configurations and their static, non-dynamic properties, every valid reachability has a formal proof.
2. Otherwise, symbolically execute $\varphi_{\text{pre}}$ for one step to $\varphi'_{\text{pre}}$. It results in a new proof obligation $\Gamma^L \vdash \varphi'_{\text{pre}} \Rightarrow \text{reach} \varphi_{\text{post}}$.

3. Continue with step (2) until a previous obligation occurs again (i.e., we notice a repetitive behavior). In this case, conclude the verification task as successful by circular reasoning.

To put it in one line:

| Deductive Verification | Reachability Proof | Symbolic Execution | Circular Reasoning |

2.3 Proof Generation for Deductive Verification

As previously mentioned, deductive verification can be specified by reachability formulas, and reachability reasoning consists of symbolic execution and circular reasoning. Symbolic execution helps to explore the program state space, while circular reasoning is used to handle repetitive behaviors. Therefore, our proof generation method also has two parts: (1) proof generation for symbolic execution and (2) a set of matching logic lemmas for circular reasoning.

2.3.1 Proof Generation for Symbolic Execution

Unlike concrete execution, symbolic execution causes the execution traces to branch, so it takes more care to handle. As an example, consider the following program configuration with a symbolic Boolean value $b$:

$$t(b) \equiv \langle \left( \text{if } (b) \text{ then } s_1 \text{ else } s_2 \right) \rangle_T$$

The above is the $\mathcal{K}$ syntax for writing program configurations (see Remark 1). For now, we only need to know that $t(b)$ is a symbolic configuration that consists of an if-statement, whose condition is an identifier $b$ that is mapped to a symbolic value $b$ in the program state. Therefore, $t(b)$ has two possible next configurations, depending on whether $b$ is true or false:

$$t_1 \equiv \langle (s_1) \rangle_T \text{ if } b \text{ is true} \quad t_2 \equiv \langle (s_2) \rangle_T \text{ if } b \text{ is false}$$

In our approach, the symbolic execution step of $t(b)$ is then encoded as follows:

$$\Gamma^L \vdash t(b) \Rightarrow_{\text{exec}} (t_1 \land b = \text{true}) \lor (t_2 \land b = \text{false}) \quad (5)$$

where $\Gamma^L$ is the matching logic theory of the language semantics and “$\Rightarrow_{\text{exec}}$” represents program execution, the same as in Equation (1). Formulas $t_1 \land b = \text{true}$ and $t_2 \land b = \text{false}$ are called constrained terms. A constrained term is a conjunction of a symbolic program configuration, such as $t_1$, and a path condition, such as $b = \text{true}$.

Generally speaking, symbolic execution is encoded in matching logic as the following proof:

$$\Gamma^L \vdash t \Rightarrow_{\text{exec}} (t_1 \land p_1) \lor \cdots \lor (t_n \land p_n) \lor (t \land q) \quad (6)$$

where $q \equiv \neg (p_1 \lor \cdots \lor p_n)$. Each $t_i \land p_i$ represents a possible next configuration of $t$. The last term $t \land q$ is called the remainder, where $q$ is the complement of all the other path conditions. Intuitively, $q$ states that no execution steps can be made, and thus the execution gets stuck at $t$.

2.3.2 Lemmas for Circular Reasoning

In reachability-based deductive verification, circular reasoning can be illustrated by the following proof step:

$$\Gamma^L \vdash \varphi_{\text{pre}} \Rightarrow_{\text{exec}} \varphi'_{\text{pre}} \quad \Gamma^L \cup \{ \varphi_{\text{pre}} \Rightarrow \text{reach} \varphi_{\text{post}} \} \vdash \varphi'_{\text{pre}} \Rightarrow \text{reach} \varphi_{\text{post}} \quad (7)$$

\hspace{1cm} \text{It is not the actual (Circularity) proof rule, but it gives a clear and precise intuition about how circular reasoning is carried out in formal verification. We present and discuss the actual proof rules in Section 2.}
Intuitively, to prove $\varphi_{pre} \Rightarrow reach \varphi_{post}$, we symbolically execute $\varphi_{pre}$ for one or more steps (represented by $\Rightarrow^{\text{exec}}_{+}$) to $\varphi'_{pre}$. Then, we prove $\varphi'_{pre} \Rightarrow reach \varphi_{post}$, with one additional axiom that is the original claim. In other words, circular reasoning allows us to prove the new sub-claims generated by symbolic execution as if the original claim holds. Usually, $\varphi_{pre} \Rightarrow reach \varphi_{post}$ is a so-called invariant claim, where $\varphi_{pre}$ consists of program code that has repetitive behaviors, such as a while-loop. The symbolic execution step, $\varphi_{pre} \Rightarrow^{\text{exec}}_{+} \varphi'_{pre}$, represents the unfolding of the while-loop. When the body of the loop is executed, $\varphi'_{pre}$ contains the same while-loop as $\varphi_{pre}$. Therefore, the new sub-claim about $\varphi'_{pre}$ can be resolved using the original claim about $\varphi_{pre}$.

Equation (7) is just an intuitively illustration of circular reasoning. It is not an actual reachability proof rule. The actual proof rule is (Circularity), which is explained in detail in Section 5 (Circularity) among with other reachability proof rules are first proposed in [41] and later presented as derived lemmas of matching logic in [8].

In our work, we formally encode these reachability lemmas in our proof objects and work out their detailed, machine-checkable proofs. The result is a comprehensive 4,000-line database of 191 lemmas for program verification [2], upon which our proof generation method is build.

An interesting technical aspect of proving the reachability lemmas is the handling of fixpoints and fixpoint reasoning, using the (Knaster-Tarski) proof rule in the matching logic proof system (Figure 3). Indeed, the reachability relation $\Rightarrow_{reach}$ is defined as a greatest fixpoint, and circular reasoning as shown in Equation (7) is nothing but a result of applying fixpoint reasoning to reachability (see Section 5.2).

It is interesting to remark that when we worked on proving the reachability lemmas, we discovered a bug in [14, Fig. 6(c)], where the authors applied two proof rules in the wrong order, making the entire proof invalid. It shows how complex and error-prone reachability proofs and circular reasoning can be. Thus, only by generating rigorous, machine-checkable proof objects can we trust any formal verification tools.

### 2.4 A Trustworthy Proof Checker

Our proof objects can be directly checked by Metamath [36]—a small and fast third-party proof framework. Metamath provides a tiny formal language that allows us to define the syntax and proof rules of matching logic, and to encode matching logic proofs into machine-checkable proof objects. Metamath has many independently-developed proof verifiers that can check the integrity of the encoded proof objects. Thanks to the simplicity of the Metamath language, these proof verifiers are small; some have only a few hundreds lines of code [32].

Our proof objects for formal verification are based on a small trust base, which consists of the Metamath proof framework and a 240-line formalization of matching logic in Metamath. The 240-line formalization, which we discuss in more detail in Section 4.2, defines the syntax and proof system of matching logic. It was first developed in [5] for the generation of the proof objects for concrete program execution. In this work, we take the 240-line formalization as is and build directly upon it our proof objects for formal verification.

### 2.5 Summary

$\mathcal{K}$ is a language framework that automatically generates deductive program verifiers from formal language semantics. To establish the trustworthiness of these autogenerated verifiers, we generate rigorous and machine-checkable proof objects as correctness certificates, which have a small trust base and can be directly checked by the third-party proof framework Metamath. The key characteristics of our approach are the following:

- **Trustworthiness.** Everything is reduced to matching logic reasoning and then encoded as proof objects, which can be directly proof-checked by Metamath.

- **Practicality.** Proof objects are generated for every verification tasks on a case-by-case basis, avoiding proving the correctness of the entire verifiers and/or $\mathcal{K}$, which is much harder.

---

3 We discuss more detail in Appendix E.
Figure 2: The complete formal semantics of an imperative language IMP, defined in K

- **Language-independence.** Proof generation is not specific to any programming languages or verification tasks, and is therefore language-independent, giving us trustworthy verifiers for all languages.

- **Efficiency.** On our benchmarks, both proof generation and proof checking are efficient. Proof generation usually takes several minutes, and proof checking takes a few seconds. Unlike program execution, formal verification does not involve a lot of execution steps, making the sizes of our proof objects more manageable.

3 K Framework and Formal Verification

We give an overview of K and its formal verification tool.

3.1 K Overview

K is an attempt to realize the ideal language framework vision as shown in Figure 1. Roughly speaking, K is a meta-language to define programming languages. For example, Figure 2 is the K definition of an imperative language named IMP, in only 39 lines of code. From this 39-line language semantics, K is able to generate all language tools for IMP, including a parser, an interpreter for program execution, and a deductive verifier for formal verification.

The K semantics for IMP in Figure 2 consists of two modules: a syntax module on the left and a semantics module on the right. The syntax module, named IMP-SYNTAX, uses the conventional BNF grammar to define the language syntax in terms of production rules. For example, lines 11-12 define if-statements. Production rules can take additional attributes that carry either syntactic or semantic information. The if-statements have attribute [strict(1)], which carries semantic information and means that the evaluation order of if-statements is strict in the first argument, i.e., in the conditions. Other attributes carry syntactic information and only affects parsing. For example, attribute [left] in line 6 states that the expression addition construct “+” is left-associative.

The semantics module, named IMP, defines program configurations as well as formal semantics. K uses configurations to organize all the semantic information that is needed to execute programs. For IMP, configurations are simple, consisting of two parts (lines 23-25): a piece of program code to be executed and
a program state that maps identifiers to their values. Configurations are organized into cells, which are denoted in an XML-like syntax in K. In IMP, <k/> is the cell of program code and <state/> is the cell of program states. Both are included in the top-level cell <T/>.

The semantics module IMP defines an operational semantics using rewrite rules. For example, the following are the rules that define the semantics of if-statements (lines 33-34):

```plaintext
rule if (I) S_ => S requires I /= Int 0
rule if (0) _ S => S
```

The first rule rewrites the statement to the then-branch, if the condition is nonzero. The second rule, on the other hand, rewrites it to the else-branch.

As another example, the following is the semantics of identifier/variable lookup (lines 26-27):

```plaintext
rule <k> X:Id => I ...
<k>
<state>...
X |- I ...
</state>
```

where we get an identifier X in the <k/> cell and look up its value I in the <state/> cell, by matching on the identifier-value binding X↦→ I. Then, we rewrite X to I and finish the lookup. These rewrite rules are similar to those in rewrite engines, such as Maude [10].

3.2 Language Tools Generated by K

We use a simple example to illustrate the execution and verification tools generated by K.

3.2.1 Concrete Execution

Let us consider the following sum100 program:

```plaintext
int n, s; n = 100; s = 0;
while (!(n <= 0)) { s = s + n; n = n + -1; }
```

The sum100 program computes the sum from 1 to 100 and stores the result 1 + 2 + ··· + 100 = 5050 in the identifier s. To execute this concrete program, K first puts it in the <k/> cell to form the following initial configuration:

```plaintext
<T> <k> sum100 </k> <state> .Map </state> </T>
```

where .Map denotes the empty map. Then, K applies the rewrite rules in Figure 2 to rewrite the above configuration until termination, and returns the final configuration:

```plaintext
<T> <k> . </k> <k> n |-> 0 , s |-> 5050 </k> </T>
```

where <k> . </k> denotes the empty computation. The value of s is 5050, as expected.

Remark 1. For better readability, we prefer to write K configurations in the following format:

```plaintext
⟨ ⟨sum100⟩k ⟨.Map⟩state ⟩T  and  ⟨ ⟨·⟩k (n → 0, s → 5050)⟩state ⟩T
```

The above are the initial and final configurations of sum100, respectively.

3.2.2 Symbolic Execution

K can execute programs with symbolic values. Consider the following sum program with a symbolic value n ≥ 0:

```plaintext
int n, s; n = n; s = 0;  // note that n is assigned to a symbolic value n
while (!(n <= 0)) { s = s + n; n = n + -1; }
```
Same as concrete execution, $K$ first puts $\sum$ in the $\langle k/\rangle$ cell and forms the *symbolic* configuration:

$$\langle \langle \sum \rangle_k \langle \cdot, \text{Map} \rangle_{\text{state}} \rangle_T \equiv \langle \langle \text{int}, n; s; n = n; s = 0; \text{while} \ldots \rangle_k \langle \cdot, \text{Map} \rangle_{\text{state}} \rangle_T$$

Depending on the value of $n$, the above configuration has many possible execution traces. If $n = 0$, the execution trace does not enter the white-loop. If $n = 1$, it enters the white-loop once. If $n = 2$, twice, and so on. It is not possible to compute all of the infinitely many final configurations.

Thus, we instrument $K$ and make it compute configurations in a *bounded* number of steps. For example, by setting the appropriate bound such that $K$ can enter the white-loop at most twice, we can obtain the following configurations and their path conditions, represented as a disjunction:

$$
\langle \langle \cdot \rangle_k \langle n \mapsto 0 \land s \mapsto 0 \rangle_{\text{state}} \rangle_T \land n = 0 \\
\lor \langle \langle \cdot \rangle_k \langle n \mapsto 0 \land s \mapsto 1 \rangle_{\text{state}} \rangle_T \land n = 1 \\
\lor \langle \langle \cdot \rangle_k \langle n \mapsto 0 \land s \mapsto 3 \rangle_{\text{state}} \rangle_T \land n = 2 \\
\lor \langle \langle \text{while} \ldots \rangle_k \langle n \mapsto n - 2 \land s \mapsto n + (n - 1) \rangle_{\text{state}} \rangle_T \land n \geq 3
$$

Symbolic execution plays an important role in our proof generation method. We instrument $K$ to generate *proof hints*, which consist of symbolic execution steps (as above), annotated with the detailed rewriting information, such as the semantic rules that are applied and the corresponding substitution results. We discuss proof hints in more detail in Section.

3.2.3 Formal Verification

A more common tool for symbolic programs is a deductive program verifier. We can use $K$ to formally verify *functional correctness* of programs. For example, the following reachability claim states that the program $\sum$, on termination, assigns the correct value $1 + 2 + \cdots + n = n(n + 1)/2$ to $s$:

$$\langle \langle \sum \rangle_k \langle \cdot, \text{Map} \rangle_{\text{state}} \rangle_T \land n \geq 0 \Rightarrow \text{reach}\langle \langle \cdot \rangle_k \langle n \mapsto 0, s \mapsto n(n + 1)/2 \rangle_{\text{state}} \rangle_T$$

As previously mentioned, the reachability relation “$\Rightarrow\text{reach}$” captures partial correctness. It is weaker than rewriting (i.e., program execution), in the sense that only divergent, finite traces are required to satisfy the right-hand side.

To help $K$ verify the above reachability claim of $\sum$, we also need to provide an *invariant* that summarizes the behaviors of the white-loop in $\sum$. The invariant is also specified as a reachability claim:

$$\forall u. \forall v. \langle \langle \text{while} \ldots \rangle_k \langle n \mapsto u, s \mapsto v \rangle_{\text{state}} \rangle_T \land u \geq 0 \\
\Rightarrow \text{reach}\langle \langle \cdot \rangle_k \langle n \mapsto 0, s \mapsto v + u(u + 1)/2 \rangle_{\text{state}} \rangle_T$$

Given the original correctness claim and the above invariant claim, $K$ automatically verifies both claims via reachability reasoning. The latter is explained in detail in Section.

To conclude, $K$ is a language framework that allows language designers to define the formal semantics of their languages. Execution and formal verification tools are automatically generated from the formal semantics, in a language-independent manner.

4 Matching Logic and Its Formalization

Matching logic is the logical foundation of $K$. Every $K$ formal semantics is compiled into a matching logic theory. Each language task that $K$ performs (program execution, formal verification, etc.) is specified by a matching logic formula. The correctness of $K$ is justified by matching logic proofs.

---

4We simplify the output for readability. The actual output often consists of non-simplified path conditions. For example, instead of $n = 1$, it uses $n \neq 0 \land n - 1 = 0$.

5The actual ANSI encoding of the invariant and the original reachability claim in $K$ can be found in Appendix.
In this section, we present the basics of matching logic. We introduce the syntax and proof system of matching logic. Then, we review the 240-line formalization of matching logic in Metamath, which forms the small trust base of our proof objects. Finally, we present some important matching logic theories, especially the theory of rewriting and program execution.

4.1 Matching Logic Overview

Matching logic was first proposed in [43] as a means to specify and reason about programs compactly and modularly. It was developed in a series of work [40, 8, 9] and was finalized in [7]. Matching logic has been adopted as the logical foundation of K [5].

Matching logic is both simple and expressive. It has 8 syntactic constructs (Definition 1) and a small proof system with 15 proof rules (Figure 3). The simplicity of matching logic makes it possible to be fully defined in only 240 lines of code in the proof framework Metamath. On the other hand, matching logic is highly expressive. Complex concepts such as sorts and rewriting can be defined using symbols and axioms, which form logical theories. Many well-known logics and calculi have been defined in matching logic, including FOL, separation logic, modal logic, Hoare logic, and type systems. In this paper, we focus on matching logic theories that define language semantics in K.

4.1.1 Matching Logic Syntax

Matching logic formulas are called pattern. Here, we define the syntax of patterns. Let us fix two sets of variables $EV$ and $SV$. We use metavariables $x,y,z,...$ to range over $EV$, and $X,Y,Z,...$ to range over $SV$. Then, the syntax of patterns is defined as follows:

**Definition 1.** A (matching logic) signature $\Sigma$ is a set of (constant) symbols. The set of $\Sigma$-patterns, or simply patterns, is inductively defined by the following grammar, with 8 syntactic constructs:

$$
\varphi ::= x \in EV \mid X \in SV \mid \sigma \in \Sigma \mid \varphi_1 \varphi_2 \mid \bot \mid \varphi_1 \rightarrow \varphi_2 \mid \exists x. \varphi \mid \mu X. \varphi
$$

where in the least fixpoint pattern $\mu X. \varphi$, we require that $\varphi$ has no negative occurrences of $X$.

According to Definition 1, element variables, set variables, and symbols are patterns. $\varphi_1 \varphi_2$ is a pattern, called application, where the first argument is applied to the second. In addition, we have propositional connectives $\bot$ and $\varphi_1 \rightarrow \varphi_2$, existential quantification $\exists x. \varphi$, and least fixpoints $\mu X. \varphi$. From the above basic syntax, we define the following derived constructs as notations, in the usual way:

- $\neg \varphi \equiv \varphi \rightarrow \bot$
- $\top \equiv \neg \bot$
- $\varphi_1 \land \varphi_2 \equiv (\neg \varphi_1 \vee \neg \varphi_2)$
- $\forall x. \varphi \equiv \neg \exists x. \neg \varphi$
- $\nu X. \varphi \equiv \neg \mu X. \neg \varphi[X/X]$

Following the convention, we denote the free variables in a pattern $\varphi$ by $\text{FreeVars}(\varphi)$. We denote capture-avoiding substitution by $\varphi[v/x]$ and $\varphi[v/X]$. They are the results of substituting $v$ for $x$ (resp. $X$) in $\varphi$, where bound variables are automatically renamed to avoid variable capture. Their formal definitions can be found in Appendix A.

4.1.2 Matching Logic Semantics

We do not need the semantics of matching logic in this paper, so we exile it to Appendix B and only give some intuition here. In short, matching logic has a pattern matching semantics. Given a model $M$ and a variable valuation $\rho$, a pattern $\varphi$ is evaluated to $|\varphi|_{M,\rho}$, which is a set of elements in $M$ that match $\varphi$. For example, $|\varphi_1 \land \varphi_2|_{M,\rho}$ is the intersection $|\varphi_1|_{M,\rho} \cap |\varphi_2|_{M,\rho}$; $|\varphi_1 \lor \varphi_2|_{M,\rho}$ is the union $|\varphi_1|_{M,\rho} \cup |\varphi_2|_{M,\rho}$; and $|\neg \varphi|_{M,\rho}$ is the complement $M \setminus |\varphi|_{M,\rho}$.
4.1.3 Matching Logic Proof System

We show the Hilbert-style matching logic proof system in Figure 3. The proof system defines the provability relation $\Gamma \vdash \varphi$, which means that $\varphi$ is provable using the proof system, with patterns in $\Gamma$ being regarded as additional axioms. We call $\Gamma$ a matching logic theory.

The matching logic proof system plays an important role in our proof generation method so we discuss it in detail. First, we need the following notion of application contexts, used in Figure 3.

**Definition 2.** An application context is a pattern $C$ with a hole variable $\square$ such that

1. $C \equiv \square$ is the identity context; or
2. $C \equiv \varphi C'$ or $C \equiv C' \varphi$, where $C'$ is an application context and $\square \not\in \text{FreeVars}(\varphi)$.

In other words, $C$ is an application context if there are only applications from the root of $C$ to $\square$.

As shown in Figure 3, the proof system has 15 proof rules, divided into 4 categories: FOL reasoning, frame reasoning, fixpoint reasoning, and some technical rules. For FOL reasoning, we have the complete proof rules for FOL (see, e.g., [44]). For frame reasoning, we state that application contexts are commutative with disjunctive connectives such as $\lor$ and $\exists$. In particular, the (Frame) rule allows us to lift local reasoning $\vdash \varphi_1 \rightarrow \varphi_2$ to a larger application context $\vdash C[\varphi_1] \rightarrow C[\varphi_2]$. For fixpoint reasoning, we have the standard fixpoint proof rules as in modal $\mu$-calculus [29]. Finally, the last two technical proof rules are added for some completeness results [8, Theorem 16].

In this paper, fixpoint reasoning is particularly important because circular reasoning is a form of fixpoint reasoning. In matching logic, the least fixpoint pattern $\mu X. \varphi$ is evaluated to the smallest set $X$ such that the equation $X = \varphi$ holds (note that $\varphi$ may include recursive occurrences of $X$), and $\nu X. \varphi$ is evaluated to
the largest such set. Therefore, it is not surprising that the following proof rules are sound:

(Knaster-Tarski) \( \frac{\varphi[\psi/X] \rightarrow \psi}{\mu X. \varphi \rightarrow \psi} \)  
(\( \mu \)-Fixpoint) \( \frac{\mu X. \varphi \leftrightarrow \varphi[(\mu X. \varphi)/X]}{\mu X. \varphi \rightarrow \psi} \)

(Knaster-Tarski\( _{\nu} \)) \( \frac{\psi \rightarrow \nu X. \varphi}{\nu X. \varphi \leftrightarrow \varphi[(\nu X. \varphi)/X]} \)  
(\( \nu \)-Fixpoint) \( \frac{\nu X. \varphi \leftrightarrow \varphi[(\nu X. \varphi)/X]}{\nu X. \varphi \rightarrow \psi} \)

Intuitively, (\( \mu \)-Fixpoint) and (\( \nu \)-Fixpoint) state that \( \mu X. \varphi \) and \( \nu X. \varphi \) are indeed fixpoints. The two (Knaster Tarski) proof rules are a direct logical incarnation of the Knaster-Tarski fixpoint theorem [49], and are what support inductive/coinductive reasoning.

Circular reasoning used by K’s program verification tool is a special case of fixpoint reasoning. Any circular proofs that K carries out for formal verification can be reduced to the more basic matching logic proof rules such as (Knaster-Tarski). This way, we reduce the complex and error-prone program verification into 15 simple matching logic proof rules and the machine-checkable proof objects.

4.2 A 240-line Formalization of Matching Logic

To encode matching logic proofs into machine-checkable proof objects, we need a formalization of matching logic that defines its syntax and proof system. In this work, we reuse the 240-line formalization developed in [5].

The formalization is implemented in a proof framework called Metamath [36]. Metamath has a tiny formal language to define formal systems and encode formal proofs. The main advantage of Metamath is its simplicity and fast proof checking. In [35], the authors formalize set theory in Metamath and use it to prove thousands of mathematical theorems, constituting a comprehensive database of basic mathematical facts. In [5], the authors use Metamath to encode proof objects for program execution. The generated proof objects have millions of line of code and can be checked by Metamath in a few seconds.

Within the 240-line formalization, there is the syntax of matching logic, some metalevel operations such as free variables and capture-avoiding substitution, and the entire matching logic proof system. We put the formalization in Appendix C for reference.

Outside the 240-line formalization, there are matching logic theories that define the practical concepts and instruments using axioms. In this work, we develop the theory of reachability and formal verification.

4.3 Basic Matching Logic Theories

In matching logic, complex concepts and mathematical instruments are defined by axioms, which form a theory. Here, we review some important theories that define equality, sorts, and rewriting.

4.3.1 Theory of Equality

By equality, we mean a pattern \( \varphi_1 = \varphi_2 \) that holds (i.e., equivalent to \( \top \)) if \( \varphi_1 \) and \( \varphi_2 \) are matched by the same elements. Otherwise, it fails (i.e., equivalent to \( \bot \)). To define \( \varphi_1 = \varphi_2 \), we first define \( [\varphi] \), called definedness. Intuitively, \( [\varphi] \) states that \( \varphi \) is defined, i.e., it is matched by at least one element (i.e., it is not \( \bot \)).

**Definition 3.** Let \( [\_] \in \Sigma \) be the definedness symbol. We write \( [\varphi] \) to mean the application \( [\_] \varphi \). We define the following axiom for \( [\_] \):

\[
\text{(Definedness)} \quad \forall x. \ [x]
\]

Intuitively, (Definedness) states that any element \( x \) is defined. Indeed, according to the semantics of matching logic, \( x \) can be matched by a unique element, and thus it is not \( \bot \).

Using the definedness symbol, we can define equality as follows:

\[
\varphi_1 = \varphi_2 \equiv \neg [\neg (\varphi_1 \leftrightarrow \varphi_2)]
\]

(8)
Section 5.1 proves that the above definition gives us the desired semantics of equality. Further proves that it gives us the standard equational deduction, using the matching logic proof system and the axiom (Definedness).

4.3.2 Theory of Sorts

K uses sorts to represent the syntactic categories of a programming language. However, matching logic is unsorted. To translate K into matching logic, we need to add support for sorts and sorted operations to matching logic.

We follow the systematic paradigm developed in [7] to define sorts. The main idea is to introduce a symbol \( \mathbb{J}_K \in \Sigma \) called the inhabitant symbol. Then for each sort, say Nat for natural numbers, we define it as a symbol that represents the sort name. Its inhabitant set is represented by \( \mathbb{J}_{\text{Nat}} \), which stands for the application \( \mathbb{J}_K \text{Nat} \). To express that a pattern \( \varphi \) is well-sorted, i.e., all elements that match \( \varphi \) have a certain sort \( s \), we write \( \varphi \subseteq \mathbb{J}_s \), where \( \subseteq \) represents the subset inclusion, defined similarly to Equation (8), except that we replace \( \leftrightarrow \) by \( \rightarrow \) (see [7] for more detail).

4.3.3 Theory of Rewriting

In K, formal semantics is defined using rewrite rules. A rewrite rule \( \text{lhs} \Rightarrow \text{rhs} \) states that for any program configuration that matches \( \text{lhs} \), it can rewrite to \( \text{rhs} \) in one step. Therefore, rewrite rules define a transition relation over program configurations.

In matching logic, a transition relation can be represented by a symbol \( \bullet \in \Sigma \) called one-path next. For any configuration \( \gamma \), \( \bullet \gamma \) is matched by all configurations that can rewrite to \( \gamma \) in one step. Therefore, we can define one-step rewriting as the following notation:

\[
\varphi_1 \Rightarrow^1_{\text{exec}} \varphi_2 \equiv \varphi_1 \rightarrow \bullet \varphi_2 \quad // \text{one-step rewriting}
\]

Intuitively, the above states that for any configuration \( \gamma \) matching \( \varphi_1 \), it also matches \( \bullet \varphi_2 \), i.e., there exists \( \gamma' \) matching \( \varphi_2 \) such that \( \gamma \) rewrites to \( \gamma' \). Therefore, one-step rewriting can be used to define K rewrite rules.

Program execution is the reflexive and transitive closure of one-step rewriting. Therefore, it can be defined as follows, where \( \mu \) is the least fixpoint operation in matching logic:

\[
\diamond \varphi \equiv \mu X. \varphi \lor \bullet X \quad // \text{“eventually”}; \text{equals to } \varphi \lor \bullet \varphi \lor \bullet \bullet \varphi \lor \ldots
\]

\[
\varphi_1 \Rightarrow^{\text{exec}}_1 \varphi_2 \equiv \varphi_1 \rightarrow \diamond \varphi_2 \quad // \text{program execution (zero, one, or more steps)}
\]

\[
\varphi_1 \Rightarrow^{+}_{\text{exec}} \varphi_2 \equiv \varphi_1 \rightarrow \bullet \diamond \varphi_2 \quad // \text{program execution (one or more steps)}
\]

5 Reachability and Its Formalization

In the previous section, we have introduced matching logic and how to encode program execution as rewriting patterns. We now introduce reachability patterns that encode formal verification.

We start with an overview of reachability logic in Section 5.1. Then we show the definition of reachability formulas in matching logic in Section 5.2 and present a set of matching logic lemmas that capture the reachability proof rules and thus support program verification in Section 5.2.

5.1 Reachability Logic Overview

Reachability logic, firstly proposed in [11] and further developed in [13, 14], is a language-independent logic for reasoning about functional correctness properties of programs based on their operational semantics. Reachability logic is the logical foundation of K’s formal verification tool. The key concept in reachability logic is that of a reachability formula:

\[
\varphi_1 \Rightarrow \varphi_2
\]
In the beginning of a reachability proof, we start with a judgment such as Equation (10) with
\[ A \proves_{reach} \varphi_1 \Rightarrow \varphi_2 \]
where \( \varphi_1 \) and \( \varphi_2 \) are matching logic patterns matched by program configurations. In other words, reachability logic is built upon matching logic.

In reachability logic, we are interested in deriving reachability judgments of the following form:

\[ A \proves_{reach} \varphi_1 \Rightarrow \varphi_2 \]

where \( A \) (axioms) and \( C \) (circularities) are sets of reachability formulas. The meaning of Equation (9) is that of a partial correctness property. For example, the following reachability judgment:

\[ A \proves_{reach} \varphi_1 \Rightarrow \varphi_2 \]

means that in a programming language whose semantics is given by \( A \) (a set of reachability formulas), any configuration \( \gamma \) that matches \( \varphi_1 \) either has an infinite execution trace or eventually reaches a configuration \( \gamma' \) that matches \( \varphi_2 \).

The axiom set \( A \) and the circularity set \( C \) in Equation (9) are used and modified in reachability proofs. In Figure 4, we show the sound and relatively complete proof system for reachability logic \([41]\). For the purpose of this paper, only (Circularity) and (Transitivity) are important and we explain them using an example below. In the beginning of a reachability proof, we start with a judgment such as Equation (10) with \( C = \emptyset \). As the proof goes, new proof obligations are added to \( C \) using (Circularity) and later moved to \( A \) using (Transitivity).

We use one simple example to illustrate (Circularity) and (Transitivity). Consider a language with one constructor \( c \) and one axiom \( A = \{ c(n) \land n \geq 1 \Rightarrow c(n-1) \} \), where \( n \) is a free variable. The rewrite rule says that \( c(n) \) rewrites to \( c(n-1) \) if \( n \geq 1 \). In this semantics, any configuration \( c(n) \) with \( n \geq 1 \) will eventually reach \( c(0) \). In reachability logic, this functional correctness property is specified as \( A \proves_{reach} c(n) \land n \geq 1 \Rightarrow c(0) \).

To prove this reachability claim, we first generalize it to \( A \proves_{reach} (\exists n. c(n) \land n \geq 1) \Rightarrow c(0) \) using (Consequence). Then, we carry out circular reasoning and use (Circularity) to add the current claim to the circularity set:

\[ A \proves_{reach} (((\exists n. c(n) \land n \geq 1) \Rightarrow c(0))) \]

Note that we cannot use the circularity claim. To use it, we need to apply (Transitivity) to make at least one execution step, which moves the circularity claim to the axiom set.

To apply (Transitivity), we first apply (Abstraction) to get rid of the quantifier \( \exists n \) in the claim and perform (Case Analysis) on \( n \). If \( n = 1 \), the proof resolves by the axiom in \( A \). Otherwise, we apply (Transitivity) to push...
the circularity claim into the axiom set:

\[
\begin{align*}
(1) & \quad A \vdash^{reach} \{\exists n. c(n) \land n \geq 1\} \quad c(n) \land n \geq 2 \Rightarrow c(n-1) \land n - 1 \geq 1 \\
(2) & \quad A \cup \{\exists n. c(n) \land n \geq 1\} \vdash^{reach} c(0) \quad c(n) \land n \geq 2 \Rightarrow c(0)
\end{align*}
\]

Finally, (1) is proved by the axiom in \( A \) and (2) is proved by the circular claim \( (\exists n. c(n) \land n \geq 1) \Rightarrow c(0) \), which has been added to the axiom set.

To sum up, in reachability logic, circular proofs are carried out by (Circularity) (which creates new circularities) and (Transitivity) (which moves circularities into axioms). Combining these two proof rules, we can reason about any repetitive behaviors of programs, thanks to the relative completeness theorem [11, Theorem 2].

### 5.2 Matching Logic Theory of Reachability

Recently, it is shown that reachability logic can be fully defined as a matching logic theory [8]. Reachability formulas become matching logic patterns, and the reachability proof rules in Figure 4 can be derived using the matching logic proof system in Figure 3.

In this work, we implement the matching logic encoding of reachability formulas into our proof objects in Metamath. Specifically, we formalize 191 matching logic lemmas that are related to reachability, and the matching logic proof system in Figure 3.

Recall from Section 4.3.3 that we use the one-path next symbol \( \bullet \in \Sigma \) to define formal semantics and program execution, using the rewriting patterns. Similarly, reachability patterns are introduced to specify verification tasks:

\[
\begin{align*}
\Diamond w\varphi & \equiv \nu X. \varphi \lor \bullet X & \quad \text{// “weak eventually”; compare it to } \Diamond \varphi \equiv \mu X. \varphi \lor \bullet X \\
\varphi_1 \Rightarrow_{reach} \varphi_2 & \equiv \varphi_1 \rightarrow \Diamond w\varphi_2 & \quad \text{// reachability (zero, one, or more steps)} \\
\varphi_1 \Rightarrow^+_{reach} \varphi_2 & \equiv \varphi_1 \rightarrow \bullet \Diamond w\varphi_2 & \quad \text{// reachability (one or more steps)}
\end{align*}
\]

Compared to program execution, the definition of reachability patterns uses the greatest fixpoint operator \( \nu \) to define “weak eventually” \( \Diamond w\varphi \), whereas \( \Diamond \varphi \) (Section 4.3.3) is a least fixpoint pattern. Thus, any configurations that match (satisfy) \( \Diamond \varphi \) must also match \( \Diamond w\varphi \), as expected. On the other hand, \( \Diamond w\varphi \) may be matched by more configurations. Intuitively, they are the configurations that are not well-founded, meaning that they can have infinite execution traces. The above intuition is justified by the following result:

**Lemma 4.** \( \vdash \Diamond w\varphi = (\Diamond \varphi \lor \nu X. \bullet X) \), where \( \nu X. \bullet X \) is matched by non-well-founded configurations.

Using reachability patterns, we can encode reachability judgments into matching logic proof obligations. Specifically, the reachability judgment \( A \vdash_{reach}^C \varphi_1 \Rightarrow \varphi_2 \) is encoded as the following matching logic proof object, following [8, Section VIII]:

\[
\vdash \bigwedge_{(\psi_1 \Rightarrow \psi_2) \in A} \forall \text{FreeVars}(\psi_1) \cup \text{FreeVars}(\psi_2), \Box (\psi_1 \Rightarrow^+_{reach} \psi_2)
\]

rules in \( A \) always hold, and thus we use “\( \Box \)”

\[
\wedge \bigwedge_{(\psi_1 \Rightarrow \psi_2) \in C} \forall \text{FreeVars}(\psi_1) \cup \text{FreeVars}(\psi_2), \Diamond (\psi_1 \Rightarrow^+_{reach} \psi_2) \rightarrow (\varphi_1 \Rightarrow_{reach} \varphi_2)
\]

rules in \( C \) are circularities and hold if any step is made, so we use “\( \Diamond \)”

where \( \Rightarrow \) is “\( \Rightarrow \) if \( C \) is nonempty and “nothing” otherwise. The operators “\( \Box \)” and “\( \Diamond \)” are defined as

\[
\begin{align*}
\Diamond \varphi & \equiv \neg \bullet \neg \varphi & \quad \text{// “all-path next”} \\
\Box \varphi & \equiv \nu X. \varphi \land \Diamond X & \quad \text{// “always”}
\end{align*}
\]
Here $\phi$ is matched by configurations whose successors all match $\phi$, while $\bullet \phi$ is matched by configurations which have at least one successor matching $\phi$.

Two key reachability rules, (Transitivity) and (Circularity), can be encoded in matching logic as:

Lemma 5. (Transitivity)

\[
\Gamma \vdash \Box \Phi \rightarrow (\phi_1 \Rightarrow \text{reach} \phi_2) \quad \Gamma \vdash \Box \Phi \rightarrow (\phi_2 \Rightarrow \text{reach} \phi_3)
\]

\[
\Gamma \vdash \Box \Phi \rightarrow (\phi_1 \Rightarrow \text{reach} \phi_3)
\]

Lemma 6. (Circularity)

\[
\Gamma \vdash \Box (\forall \text{FreeVars}(\phi_1) \cup \text{FreeVars}(\phi_2). \phi_1 \Rightarrow \text{reach} \phi_2) \rightarrow (\phi_1 \Rightarrow \text{reach} \phi_2)
\]

In (Transitivity), $\Phi$ is the conjunction of all circularities that are to be moved to the axiom set. These two rules capture the essence of circular reasoning and will be used in the proof generation procedure in Section 7.

Under the above encoding, all reachability proof rules in Figure 4 can be proved in matching logic and encoded as matching logic proof objects. See Appendix H for more technical detail about the encoding.

6 Proof Generation for Symbolic Execution

We discuss our proof generation method for symbolic execution, which is a main technical contribution of this paper. In Section 7, we combine the procedure in this section with that of circular reasoning to finalize the proof objects for program verification.

6.1 Problem Formulation

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

\[
A = \{ \text{lhs}_k \land q_k \Rightarrow^1_{\text{exec}} \text{rhs}_k \mid k = 1, 2, \ldots, K \}
\]

where $\text{lhs}_k$ represents the left-hand side of the rewrite rule, $\text{rhs}_k$ represents the right-hand side, and $q_k$ denotes the rewrite condition. For unconditional rules, $q_k$ is $\top$. The notation $\Rightarrow^1_{\text{exec}}$ stands for one-step execution, defined in Section 4.3.3. We use $\Gamma^L$ to denote the matching logic theory encoding the rewrite rules in $A$.

In symbolic execution, program configurations often appear with their corresponding path conditions. We represent them as $t \land p$, 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 a special type of 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 $t \land p$ and a list of final constrained terms $t_1 \land p_1, \ldots, t_n \land p_n$, which are returned by $\mathbb{K}$ as the result(s) of symbolic executing $t$ under the condition $p$. Each $t_i \land p_i$ represents one possible execution trace. Our goal is to generate a proof object for the following proof goal:

\[
\Gamma^L \vdash t \land p \Rightarrow_{\text{exec}} (t_1 \land p_1) \lor \cdots \lor (t_n \land p_n)
\]

(Symbolic Execution Goal)

6.2 Proof Hints

To help generate the proof of (Symbolic Execution Goal), we instrument $\mathbb{K}$ to output proof hints, which document in detail what semantic rules have been applied during the symbolic execution.

Specifically, a proof hint (for symbolic execution) consists of the detailed rewriting information for each symbolic execution step. For the $j^{th}$ step, the proof hint provides the following information:
• a constrained term \( t_{j, l}^{\text{hint}} \land p_{j, l}^{\text{hint}} \) before step \( j \);

• \( l_j \) constrained terms \( t_{j, 1}^{\text{hint}} \land p_{j, 1}^{\text{hint}}, \ldots, t_{j, l_j}^{\text{hint}} \land p_{j, l_j}^{\text{hint}} \) after step \( j \), where for each \( 1 \leq l \leq l_j \), the term is annotated with a rewrite rule index \( 1 \leq k_{j,l} \leq K \), and a substitution \( \theta_{j,l} \);

• an (optional) constrained term \( t_j^{\text{rem}} \land p_j^{\text{rem}} \), called the remainder of step \( j \).

Intuitively, each constrained term \( t_{j, l}^{\text{hint}} \land p_{j, l}^{\text{hint}} \) for \( 1 \leq l \leq l_j \) represents one execution branch starting at \( t_j^{\text{hint}} \land p_j^{\text{hint}} \), obtained by applying the \( k_{j,l} \)-th rewrite rule in \( A \):

\[
lhs_{k_{j,l}} \land q_{k_{j,l}} \Rightarrow_{\text{exec}} \, rhs_{k_{j,l}}
\]

with the corresponding substitution \( \theta_{k_{j,l}} \). The remainder \( t_j^{\text{rem}} \land p_j^{\text{rem}} \) denotes the branch where no rewrite rules can be applied further and thus the execution gets stuck. In theory, \( t_j^{\text{rem}} \) and \( t_j^{\text{rem}} \land p_j^{\text{rem}} \) should be equal, but in practice they may look different, because \( t_j^{\text{rem}} \land p_j^{\text{rem}} \) may be further simplified by \( K \) using the stronger condition \( p_j^{\text{rem}} \).

From the above proof hint, we generate a proof for each symbolic execution step. For example, the following is the proof for the \( j \)-th symbolic execution step:

\[
\Gamma^L \vdash (t_j^{\text{hint}} \land p_j^{\text{hint}}) \Rightarrow_{\text{exec}} (t_{j, 1}^{\text{hint}} \land p_{j, 1}^{\text{hint}}) \lor \cdots \lor (t_{j, r_j}^{\text{hint}} \land p_{j, r_j}^{\text{hint}}) \lor (t_j^{\text{rem}} \land p_j^{\text{rem}})
\]

(Step \( j \))

To prove \( \text{Step} \( j \) \), we need to generate proofs for each execution branch \( 1 \leq l \leq l_j \):

\[
\Gamma^L \vdash (t_j^{\text{hint}} \land p_j^{\text{hint}}) \Rightarrow_{\text{exec}} (t_{j, l}^{\text{hint}} \land p_{j, l}^{\text{hint}})
\]

(Branch \( j, l \))

For the remainder branch, we need to prove that:

\[
\Gamma^L \vdash (t_j^{\text{hint}} \land p_j^{\text{hint}}) \Rightarrow (t_j^{\text{rem}} \land p_j^{\text{rem}})
\]

(remainder \( j \))

6.3 Proof Generation

Given the above proof hint, we prove \( \text{(Symbolic Execution Goal)} \) in the following steps:

**Step 1.** We prove \( \text{(Branch} j, l \text{)} \) and \( \text{(Remainder} j \text{)} \) for each \( j \) and branch \( 1 \leq l \leq l_j \).

**Step 2.** We combine \( \text{(Branch} j, l \text{)} \) and \( \text{(Remainder} j \text{)} \) to obtain a proof of \( \text{(Step} j \text{)} \).

**Step 3.** We combine \( \text{(Step} j \text{)} \) to prove \( \text{(Symbolic Execution Goal)} \).

**Lemmas and Their Proofs.** We need many lemmas about program execution \( \Rightarrow_{\text{exec}} \) when we generate the proof objects for symbolic execution. The most important and relevant lemmas are stated explicitly in this paper. In total, 191 new lemmas are formally encoded, and their proofs have been completely worked out based on the 240-line Metamath database of matching logic. These lemmas can be easily reused for future development.

In the following, we explain each proof generation step in detail.

6.3.1 Step 1: Proving \( \text{(Branch} j, l \text{)} \) and \( \text{(Remainder} j \text{)} \)

Recall that \( \text{(Branch} j, l \text{)} \) is obtained by applying the following rewrite rule from the language semantics (where \( 1 \leq k_{j,l} \leq K \)):

\[
lhs_{k_{j,l}} \land q_{k_{j,l}} \Rightarrow_{\text{exec}} \, rhs_{k_{j,l}}
\]

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

\[
\Gamma^L \vdash lhs_{k_{j,l}} \theta_{j,l} \land q_{k_{j,l}} \theta_{j,l} \Rightarrow_{\text{exec}} \, rhs_{k_{j,l}} \theta_{j,l}
\]

(11)
Since the condition $q_{k,j}, \theta_{j,l}$ is a predicate on the free variables of Equation (11) and it holds on the left-hand side, it also holds on the right-hand side. Therefore, we prove that:

$$\Gamma^L \vdash lhsk_{j,l}, \theta_{j,l} \land q_{k,j}, \theta_{j,l} \Rightarrow \Gamma^L \vdash 1_{exec} \ rhsk_{j,l}, \theta_{j,l} \land q_{k,j}, \theta_{j,l}$$ (12)

To proceed the proof, we need the following lemma:

Lemma 7.

$$(\Rightarrow 1_{exec} \text{ Consequence}) \quad \Gamma^L \vdash \varphi_1 \Rightarrow \varphi_1' \quad \Gamma^L \vdash \varphi_1' \Rightarrow 1_{exec} \varphi_2' \quad \Gamma^L \vdash \varphi_2' \Rightarrow \varphi_2$$

Intuitively, Lemma 7 allows us to strengthen the left-hand side and/or weaken the right-hand side of an execution relation. Using Lemma 7 and by comparing our proof goal (Branch$_{j,l}$) with Equation (12), we only need to prove the following two implications, called subsumptions:

$$\Gamma^L \vdash (t_j^{\text{hint}} \land p_{j,l}^{\text{hint}}) \Rightarrow (lhsk_{j,l}, \theta_{k,l}, \land q_{k,j}, \theta_{k,l})$$ (left-hand side strengthening)

$$\Gamma^L \vdash (rhsk_{j,l}, \theta_{k,l}, \land q_{k,j}, \theta_{k,l}) \Rightarrow (t_j^{\text{hint}} \land p_{j,l}^{\text{hint}})$$ (right-hand side weakening)

It is common to prove subsumptions as above in our proof generation method. For example, (Remainder$_j$) is also a subsumption. We elaborate on subsumption proofs later in Section 6.3.4.

6.3.2 Step 2: Proving (Step$_j$).

The proof goal (Step$_j$) is proved by combing the following proofs for each branches and the remainder:

$$\Gamma^L \vdash 1_{exec} \ j^{\text{hint}} \land p_{j,l}^{\text{hint}} \quad (\text{Branch}_{j,l})$$

$$\vdots$$

$$\Gamma^L \vdash 1_{exec} \ j^{\text{hint}} \land p_{j,l}^{\text{hint}} \quad (\text{Branch}_{j,l})$$

$$\Gamma^L \vdash 1_{exec} \ j^{\text{hint}} \land p_{j,l}^{\text{rem}} \Rightarrow t_j^{\text{rem}} \land p_j^{\text{rem}} \quad (\text{Remainder}_{j})$$

Note that our proof goal (Step$_j$) uses "$\Rightarrow 1_{exec}$", 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 transitive-reflexive execution relation "$\Rightarrow 1_{exec}$":

Lemma 8.

$$(\Rightarrow 1_{exec} \text{ Introduction}_1) \quad \Gamma^L \vdash \varphi_1 \Rightarrow \varphi_2$$

$$(\Rightarrow 1_{exec} \text{ Introduction}_2) \quad \Gamma^L \vdash \varphi_1 \Rightarrow 1_{exec} \varphi_2$$

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:

$$\Gamma^L \vdash p_j^{\text{hint}} \Rightarrow p_j^{\text{hint}} \lor \ldots \lor p_j^{\text{hint}} \lor p_j^{\text{rem}}$$ (13)

The above implication includes only logical constraints and no configuration terms, and thus belongs to domain reasoning. Therefore, we simply translate it into an equivalent FOL formula and delegate it to SMT solvers, such as Z3 [16]. In our current proof generation method, we do not generate proofs for SMT domain reasoning (see Section 9.2.1).

From Equation (13), we can prove that the left-hand side of (Step$_j$), $j^{\text{hint}} \land p_j^{\text{hint}}$, can be broken down into $l_j + 1$ branches, according to their path conditions:

$$\Gamma^L \vdash (t_j^{\text{hint}} \land p_j^{\text{hint}}) \Rightarrow (t_j^{\text{hint}} \land p_j^{\text{hint}}) \lor \ldots \lor (t_j^{\text{hint}} \land p_j^{\text{hint}}) \lor (t_j^{\text{hint}} \land p_j^{\text{rem}})$$ (14)

Note that that right-hand side of Equation (14) is exactly the disjunction of all the left-hand sides of (Branch$_{j,l}$) and (Remainder$_j$). Therefore, to prove the proof goal (Step$_j$), we only need to prove the following lemma, which allows us to merge the executions in different branches into one:
Lemma 9.

\[(\Rightarrow_{\text{exec Merge}}) \quad \Gamma^L \vdash \varphi_1 \Rightarrow_{\text{exec}} \varphi'_1 \quad \ldots \quad \Gamma^L \vdash \varphi_n \Rightarrow_{\text{exec}} \varphi'_n\]

\[\Gamma^L \vdash \bigvee_{i=1}^{n} \varphi_i \Rightarrow_{\text{exec}} \bigvee_{i=1}^{n} \varphi'_i\]

6.3.3 Step 3: Proving (Symbolic Execution Goal).

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

\[\Gamma^L \vdash (t \land p) \Rightarrow_{\text{exec}} (t \land p)\]  \hspace{1cm} (15)

Then, we repeatedly apply the following steps to symbolically execute the right-hand side of Equation (15), until it becomes the same as the right-hand side of (Symbolic Execution Goal):

1. Suppose we have established a proof of

\[\Gamma^L \vdash (t \land p) \Rightarrow_{\text{exec}} (t^\text{im}_1 \land p^\text{im}_1) \lor \cdots \lor (t^\text{im}_m \land p^\text{im}_m)\]  \hspace{1cm} (16)

where \(t^\text{im}_1, p^\text{im}_1\), etc. represent the intermediate terms and/or constraints, respectively.

2. Look for a (Step\(\_j\)) claim of the form

\[\Gamma^L \vdash (t^\text{hint}_j \land p^\text{hint}_j) \Rightarrow_{\text{exec}} (t^\text{hint}_{j,1} \land p^\text{hint}_{j,1}) \lor \cdots \lor (t^\text{hint}_{j,l_j} \land p^\text{hint}_{j,l_j}) \lor (t^\text{rem}_j \land p^\text{rem}_j)\]  \hspace{1cm} (Step\(\_j\))

such that \(t^\text{hint}_j \land p^\text{hint}_j \equiv t^\text{im}_i \land p^\text{im}_i\), for some intermediate constrained term \(t^\text{im}_i \land p^\text{im}_i\). Without loss of generality, let us assume that \(i = 1\), i.e., it is the first intermediate constrained term \(t^\text{im}_1 \land p^\text{im}_1\) that can be rewritten/executed using (Step\(\_j\)).

3. Execute \(t^\text{im}_1 \land p^\text{im}_1\) in Equation (16) for one step using (Step\(\_j\)), and obtain the following proof:

\[\Gamma^L \vdash (t \land p) \Rightarrow_{\text{exec}} (t^\text{im}_1 \land p^\text{im}_1) \lor \cdots \lor (t^\text{im}_{j,l_j} \land p^\text{im}_{j,l_j}) \lor (t^\text{rem}_j \land p^\text{rem}_j)\]

right-hand side of (Step\(\_j\))

\[\lor (t^\text{im}_{j+1} \land p^\text{im}_{j+1}) \lor \cdots \lor (t^\text{im}_m \land p^\text{im}_m)\]

same as Equation (16)

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

6.3.4 Proving Subsumption of Constrained Terms

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

\[\Gamma^L \vdash (t \land p) \rightarrow (t' \land p')\]

We divide it into the following two sub-goals:

\[\Gamma^L \vdash p \rightarrow p'\]

\[\Gamma^L \vdash p \rightarrow (t = t')\]
procedure checkReachability(I)
for ϕ ⇒reach, ψ ∈ I do
    Q ← successors(ϕ);
    if Q = ∅ and ΓL ⊬ ϕ → ψ then fail;
while Q ≠ ∅ do
    Pop any ϕ’ from Q;
    if ΓL ⊢ ϕ’ → ψ then continue;
    Q’ ← successorsI(ϕ’);
    if Q’ = ∅ then fail;
else Q ← Q ∪ Q’;

Algorithm 1: Internal algorithm of K’s formal verification tool [14]. Here, successors(ϕ) is a set of patterns that are the results of symbolically executing ϕ for one step using the formal semantics (see Section 6.3). successorsI(ϕ) is the result of applying an invariant rule in I if any one is applicable, otherwise we let successorsI(ϕ) = successors(ϕ).

To prove the first sub-goal ΓL ⊬ p → 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 ΓL ⊬ p → (t = t’), we first try SMT solvers. If SMT solvers resolves the proof goal, then our proof generation method succeeds. Otherwise, we break down t and t’ into sub-terms. Specifically, if t ≡ f(t1, . . . , tn) and t’ ≡ f(t’1, . . . , t’n), we reduce the sub-goal into a set of goals:

ΓL ⊬ p → (t1 = t’1) . . . ΓL ⊬ p → (tn = t’n)

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

7 Proof Generation for Deductive Verification

In this section, we discuss our proof generation method for formal deductive verification of programs. As said in Section 5, our method is based on the language-independent reachability logic and its encoding in matching logic. The key step is the use of the reachability proof rule (Circularity) to carry out circular reasoning and thus to handle repetitive behavior of programs.

7.1 Overview of K’s Verification Tool

We first explain the internal algorithm of K’s formal verification tool.

Internally, K implements Algorithm 1 to automate formal verification based on reachability logic (Figure 4). The algorithm takes a set I of reachability rules, which are called the invariant rules. Then, at each step, the algorithm carries out symbolic execution and rewrites the left-hand side of the invariant rules in I, using the operational semantics of the programming language. In addition, once the first rewrite step is made, the algorithm marks the invariant rules as trusted rules and uses them as if they are also part of the formal language semantics, which carries out circular reasoning.

Algorithm 1 can be regarded as an optimization of the reachability proof rules in Figure 4, because it proves all invariants in I simultaneously. Such optimization is called set circularity, which is proved equivalent to the reachability proof rules, but it is easier to implement and to automate. Any proof using set circularity can be reduced to a proof using only the reachability proof rules [42, Lemma 5]. In our proof generation method, we make the following Assumption 4 which helps to simplify the reduction from K’s use of set circularity to the basic reachability proof rules in Figure 4. Relaxation of Assumption 4 is left as future work (see Section 9.2.2).
Assumption 1. The invariant rules in \( I \) have no mutual dependency. That is, there is no subset \( \{ \varphi_1 \Rightarrow \text{reach} \psi_1, \ldots, \varphi_n \Rightarrow \text{reach} \psi_n \} \subseteq I \) with \( n \geq 2 \) such that \( \varphi_i \Rightarrow \text{reach} \psi_i \) is used by Algorithm 1 in executing \( \varphi_{i+1} \) for all \( 1 \leq i < n \), and \( \varphi_n \Rightarrow \text{reach} \psi_n \) is used in executing \( \varphi_1 \).

7.2 Problem Formulation and Proof Hints

Proof generation for deductive verification can be stated as the following problem. Given the formal semantics \( \Gamma_L \) of a programming language and a set \( I \) of invariant rules that can be verified/proved by Algorithm 1, we need to generate matching logic proofs for:

\[
\Gamma^L \vdash \varphi \Rightarrow \text{reach} \psi \quad \text{for all} \quad \varphi \Rightarrow \text{reach} \psi \in I
\]

(Verification Goal)

We also instrument \( \mathbb{K} \), essentially Algorithm 1, and make it output proof hints. Since Algorithm 1 does nothing but symbolic execution, the proof hints are almost the same as that of symbolic execution in Section 6.2 except that the invariant rules in \( I \) can also be used as the language semantic rule (i.e., circular reasoning).

Therefore, the main challenge is to turn the implicit circular reasoning in Algorithm 1 into explicit applications of the (Circularity) proof rule in Figure 4.

7.3 Proof Generation

For each \( \varphi \Rightarrow \text{reach} \psi \in I \), the proof generation has two main steps:

1. Symbolically execute \( \varphi \) following the given proof hint, generated by \( \mathbb{K} \). Note that both the language semantic rules and the invariant rules in \( I \) can be used here for rewriting.

2. Prove that the results of the above symbolic execution are subsumed by \( \psi \).

Step (1) generates proofs for the symbolic execution steps in Algorithm 1 (i.e., the calls to the successors method), and Step (2) proves the subsumptions (i.e., the checks \( \Gamma^L \vdash \varphi \Rightarrow \psi \)).

We first describe Step (2) since it is simpler. Suppose we have obtained the following matching logic proof from Step (1):

\[
\Gamma^L \vdash \varphi \Rightarrow \text{reach} \psi_1 \lor \cdots \lor \psi_m
\]

where \( \psi_1, \ldots, \psi_m \) are the constrained terms for different branches, starting at \( \varphi \). Then in Step (2), to connect Equation (17) and (Verification Goal), we only need to prove the following \( m \) subsumption goals, using the method in Section 6.3.4:

\[
\Gamma^L \vdash \psi_1 \Rightarrow \psi \quad \cdots \quad \Gamma^L \vdash \psi_m \Rightarrow \psi
\]

And thus we finish the proof of (Verification Goal) successfully.

It is slightly more challenging to generate proofs of symbolic execution for Step (1) because it involves circular reasoning, which corresponds to the calls to successors method in Algorithm 1. Depending on what semantic/invariant rules are applied, there are three cases:

(A) We apply a language semantic rule.

(B) We apply an invariant rule in \( I \) that has already been proved.

(C) We apply an invariant rule that is the current proof goal \( \varphi \Rightarrow \text{reach} \psi \) as in (Verification Goal). According to Algorithm 1, it can only happen if this execution step is not the first step.

\[\text{Without Assumption 1, there can be a fourth case, where a symbolic execution step is made by applying an invariant rule that has not been proved and is also not the current proof goal. It happens when two or more invariant rules create mutual dependency, where the proof of one invariant uses the other invariants, and vice versa. Our current proof generation method does not handle such mutually dependent invariant rules, see Section 9.2.2 for more discussion.}\]
The proofs for cases (A) and (B) are identical to the usual symbolic execution proofs in Section 6.3. Therefore, we only need to handle (C), which requires (Transitivity) and (Circularity) from Section 5.2.

Since the symbolic execution step made in case (C) is not the first execution step, we have already constructed a proof of the following form, which specifies the intermediate execution result from $\varphi$:

$$\Gamma^L \vdash \varphi \Rightarrow \varphi^{im} \lor \Phi^{im}$$  \hspace{1cm} (18)

where $\varphi^{im}$ is the intermediate constrained term, whose next execution step, according to the proof hint, is made by applying the current proof goal $\varphi \Rightarrow \text{reach } \psi$; and $\Phi^{im}$ is a disjunction of the remaining intermediate terms. From the proof hint, we also know that the execution step from $\varphi^{im}$ has substitution $\theta^{im}$, and the next pattern is denoted by $\varphi^{next}$.

The proof for case (C) is then stated as the following forward proof, starting from Equation (18) (where the current circularity claim is $\Psi \equiv \forall \text{FreeVars}(\varphi) \cup \text{FreeVars}(\psi). \varphi \Rightarrow \text{reach } \psi$):

$$\Gamma^L \vdash \square \Psi \rightarrow (\varphi \Rightarrow \varphi^{im} \lor \Phi^{im})$$  \hspace{1cm} (19)

$$\Gamma^L \vdash \Psi \rightarrow (\varphi^{im} \Rightarrow \text{reach } \varphi^{next})$$  \hspace{1cm} By weakening (18)  \hspace{1cm} (20)

$$\Gamma^L \vdash \square \Psi \rightarrow (\varphi^{im} \lor \Phi^{im} \Rightarrow \text{reach } \varphi^{next} \lor \Phi^{im})$$  \hspace{1cm} By (20) and $\Gamma^L \vdash \square \varphi \rightarrow \varphi$  \hspace{1cm} (21)

$$\Gamma^L \vdash \square \Psi \rightarrow (\varphi \Rightarrow \varphi^{next} \lor \Phi^{im})$$  \hspace{1cm} By (21) and $\Gamma^L \vdash \Phi^{im} \Rightarrow \text{reach } \Phi^{im}$  \hspace{1cm} (22)

$$\Gamma^L \vdash \square \Psi \rightarrow (\varphi \Rightarrow \varphi^{next} \lor \Phi^{im})$$  \hspace{1cm} By (19), (22), and (Transitivity) in Section 5.2  \hspace{1cm} (23)

Then we continue the symbolic execution of the right-hand side of Equation (23), i.e., $\varphi^{next} \lor \Phi^{im}$. Note that after case (C), we have an extra premise $\square \Psi$, so the lemmas used in Section 6.3 need to be suitably weakened to accommodate for this premise.

After the complete symbolic execution Step (1) and Step (2), we obtain the proof of

$$\Gamma^L \vdash \square (\forall \text{FreeVars}(\varphi) \cup \text{FreeVars}(\psi). \varphi \Rightarrow \text{reach } \psi) \rightarrow (\varphi \Rightarrow \text{reach } \psi)$$

Finally, by (Circularity) in Section 5.2 we prove $\Gamma^L \vdash \varphi \Rightarrow \text{reach } \psi$, which is the desired (Verification Goal).

8 Evaluation

We evaluate our proof generation method on three different programming languages:

- IMP, as defined in Figure 2, is an imperative language with integers, Booleans, assignments, conditional statements, and while-loops;
- REG is a register-based virtual machine with a fixed number of registers;
- PCF, or programming computable functions [39], is a functional language.

As for verification examples, we consider the following three simple programs:

- SUM, which computes the sum $1 + \cdots + n$ of input $n$;
- EXP, which computes the exponential $n^k$ of inputs $n$ and $k$;
- COLLATZ, which computes the Collatz sequence [21] from input $n$ until it reaches 1.

Each program is written in three languages, so we have in total 9 verification tasks. For COLLATZ, we consider the reachability property that states that either 1 is reached in finite steps or the program loops forever. The detailed encoding of these verification tasks in $\text{K}$ are in Appendix D.

We measured the performance of both proof generation and proof checking. Experimental results are shown in Figure 5. We used a machine with Intel i7-8550U processors and 16 GB of RAM.

The main takeaways are the following:
Figure 5: Performance of our proof generation method. From left to right, we list the verification tasks, proof generation time, number of symbolic execution steps during verification, proof hint size, proof object size, and proof checking time (only including lemmas specific to the verification tasks).

1. Proof checking is fast and takes a few seconds;
2. Proof generation takes longer time, often in the order of minutes, depending on the number of symbolic execution steps involved in formal verification;
3. Proof objects are large and take hundreds of megabytes space to store.

We evaluate the experimental results in the following.

**Proof Generation.** At a high level, proof generation time consists of two phases: (1) the time to generate the formal semantics $\Gamma^L$ from the $\mathcal{K}$ language definitions, and (2) the time to generate the proof objects following the algorithms in Sections 6 and 7. In our experiments, the first phase is efficient and only takes a few seconds, which is roughly linear to the size of the $\mathcal{K}$ language definitions (i.e., the number of formal semantic rules). The second generation phase takes most of the time. It is linear to the number of symbolic execution steps that $\mathcal{K}$ has carried out during verification (i.e., the number of (Step $j$) claims) and the size of intermediate configurations.

It is worth mentioning that although proof generation takes significant amount of time, deductive verifiers are slow in general, and it takes even more time to infer the right invariant rules for a verification task and to work out full detail. Therefore, we argue that it is worth spending the extra time in generating rigorous and machine-checkable mathematical proof objects for program verifiers, and establishing the correctness of verification results on a small trust base (Section 4.2).

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

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

9 Implementation, Limitations, and Future Directions

We discuss some interesting technical details about our current proof generation implementation, as well as its limitations and future directions.
9.1 Our Current Implementation

At a high level, our implementation is two-layered. Firstly, it consists of the proof generation algorithm described in Sections 6 and 7. The main algorithms are implemented in Python, which take proof hints as input and emit complete matching logic proof objects in Metamath. The proof objects can be directly proof-checked by Metamath verifiers.

Secondly, the implementation also consists of 191 new lemmas about rewriting and/or reachability, whose formal proofs (of roughly 4,000 LOC in Metamath) have been fully, manually worked out and added to the existing Metamath database of matching logic.

9.2 Limitations and Future Directions

Our current proof generation implementation is still preliminary. We discuss some important limitations and how we plan to address them in the future.

9.2.1 Need to Trust SMT Solvers

Our current implementation delegates domain reasoning to SMT solvers and does not generate proof objects for them. By domain reasoning, we mean \( \Gamma \vdash \varphi \rightarrow \psi \), where \( \varphi \) and \( \psi \) are logical constraints about domain values, such as integers. To prove such domain properties, we encode them as FOL queries and call SMT solvers. It thus creates a gap in our proof objects, because we also need to trust the external SMT solvers, besides the 240-line formalization.

To eliminate SMT solvers from the trust base, we should generate complete proofs for domain reasoning. There has been existing research on generating proof objects for SMT solvers, such as [48, 3]. We should incorporate it in our proof generation method.

9.2.2 Set Circularity

As discussed in Section 7.1, \( \mathbb{K} \)'s formal verification tool implements Algorithm 1 as an optimized way to carry out reachability reasoning (Figure 4) efficiently. The main difference is that Figure 4 has only one proof rule (Circularity) that handles one circular/invariant rule at a time, while Algorithm 1 proves a set \( I \) of invariant/circular rules simultaneously.

Such optimization is known in literature as set circularity [42]. It is an equivalent but optimized way to do reachability proofs. Set circularity can be stated as the following rule:

\[
\frac{A \mid \text{reach} \varphi \Rightarrow \psi \text{ for all } (\varphi \Rightarrow \psi) \in I}{A \mid \text{reach} \varphi \Rightarrow \psi \text{ for all } (\varphi \Rightarrow \psi) \in I}
\]

The (Circularity) proof rule in Figure 4 can be considered a special case of (Set Circularity) where \( I \) contains only one circular rule.

Our current proof generation method supports (Set Circularity) under Assumption 1, which states that \( I \) does not include mutual dependency among the invariant rules. Under that assumption, it is easier to translate proofs using (Set Circularity) into proofs using only the basic (Circularity) proof rule. We already explained the translation in Section 7.3.

However, Assumption 1 is not necessary in theory. [42 Lemma 5] proves that a proof using (Set Circularity) can always be reduced to a proof using only (Circularity). It is left as future work to fully implement that translation in our proof generation method, and we explain some technical details in Appendix F.

9.2.3 Verification of Nondeterministic Programs

In this work, we consider only one-path reachability. In other words, \( \varphi \Rightarrow \text{reach} \psi \) holds if \( \varphi \) diverges or has one finite execution that reaches \( \psi \). While one-path reachability is sufficient for the verification of deterministic
programs, it does not work well on nondeterministic or concurrent programs: \( \psi \) might not be reachable on other finite execution traces from \( \phi \).

To address the verification of nondeterministic programs, all-path reachability was proposed [13]. An all-path reachability \( \phi \Rightarrow^\forall_{\text{reach}} \psi \) claim holds iff all finite (and maximal) execution traces can reach \( \psi \). Therefore, it supports the verification of nondeterministic programs. On deterministic programs, all-path reachability and one-path reachability coincide.

We plan to extend our proof generation method to support all-path reachability. The key is to add the following (Step) axiom, which introduces all-path claims from (one-path) rewrite rules in the semantics \( A = \{ \text{lhs}_1 \Rightarrow \text{rhs}_1, \ldots, \text{lhs}_K \Rightarrow \text{rhs}_K \} \):

\[
\text{(Step)} \quad A \vdash^\forall_{\text{reach}} \phi \Rightarrow^\forall_{\text{reach}} (\psi_1 \lor \cdots \lor \psi_K)
\]

where for \( 1 \leq k \leq K \), \( \psi_k \) is the result of executing \( \phi \) for one step, using the \( k \)-th semantic rule \( \text{lhs}_k \Rightarrow \text{rhs}_k \). Intuitively, the (Step) axiom states that an execution step must be made using one of the semantic rules in \( A \).

9.2.4 Smaller Proof Objects.

Our proof objects are based on a very small trust base. As a trade-off, the sizes of proof objects are large. In Figure 5 proof objects can take hundreds of megabytes to store. It is primarily due to the space-inefficient Metamath encoding of matching logic proofs.

Proof compression helps to greatly reduce the size of our proof objects. In a preliminary experiment, we used the generic, lossless compression tool xz [50] to compress the proof objects. For proof checking, the proofs are decompressed incrementally on-the-fly, and passed to a Metamath verifier that implements an online proof checking algorithm. This way, we can reduce the sizes of proof objects by 99% on average. For example, the size of the proof for collatz.reg is reduces from 2715 MB in Figure 5 to 9 MB. Full experimental details are presented in Appendix C.

10 Related Work and Conclusion

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

For the first approach, theorem provers such as Coq [34] and Isabelle/HOL [27] are often used to formalize language tools and prove their correctness. For example, CompCert [31] is a C compiler that is implemented and verified in Coq. CakeML [30] is an implementation of Standard ML [23], verified in HOL4 [45]. It takes great effort in verifying such systems, but when it is done, it gives a uniform guarantee on the correctness of the entire tool.

However, it is worth noting that the theorem provers used to formalize and verify language tools are themselves intricate, based on complex logical foundations. Therefore, their trust base can be large. For example, Coq is based on CIC, or calculus of inductive constructors [11], which is arguably more complex than matching logic, the logical foundation of our proof generation method: the trusted Coq kernel has nearly 25,000 lines of OCaml [12].

Recent research has been trying to reduce the trust base of theorem provers. For example, [47, 46] attempt to reduce the trust base of Coq by formalizing the kernel of Coq within Coq. [18] proposes an alternative type theory to CIC that has a smaller trust base. [24] formalizes the OCaml kernel of HOL light within HOL light itself.

The second approach, which is undertaken by our proof generation method, is to generate correctness certificates on a case-by-case basis. There has been work to generate proofs for decision procedures in SMT solvers to justify their correctness [18, 3, 37] and variants of the Edinburgh LF [22] are often used to encode
axioms and formal proofs. Although similar in spirit, our work is the first (to the authors’ knowledge) to apply this approach to a language framework and program verification.

**Conclusion**  We propose a proof-based approach to verify the results of formal verification and symbolic execution of programs in a language-independent manner. We base our approach on the K framework and its logical foundation, matching logic. For every verification task that K performs, we generate rigorous and machine-checkable matching logic proofs, which can be proof-checked based on a 240-line formalization of matching logic in Metamath. Our experiment shows promising performance of both proof generation and proof checking.

**References**


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


[30] Ramana Kumar, Magnus O Myreen, Michael Norrish, and Scott Owens. Cakeml: a verified implementa-


A Free Variables and Substitution in Matching Logic

We define the set of free variables in a matching logic pattern \( \text{FreeVars}(\varphi) \) inductively as

\[
\begin{align*}
\text{FreeVars}(x) &= \{x\} \\
\text{FreeVars}(X) &= \{X\} \\
\text{FreeVars}(\sigma) &= \text{FreeVars}(\bot) = \emptyset \\
\text{FreeVars}(\varphi_1 \to \varphi_2) &= \text{FreeVars}(\varphi_1) \cup \text{FreeVars}(\varphi_2) \\
\text{FreeVars}(\exists x. \varphi) &= \text{FreeVars}(\varphi) \setminus \{x\} \\
\text{FreeVars}(\mu X. \varphi) &= \text{FreeVars}(\varphi) \setminus \{X\}
\end{align*}
\]

Recall that lowercase metavariables such as \( X, Y, Z \) denotes set variables. The capture-avoiding substitution for element variable \( x \) defined as

\[
\begin{align*}
x[\psi/x] &\equiv \psi \\
y[\psi/x] &\equiv y \text{ if } y \neq x \\
X[\psi/x] &\equiv X \\
\bot[\psi/x] &\equiv \bot \\
\sigma[\psi/x] &\equiv \sigma
\end{align*}
\]

For set variables, \( \varphi[\psi/X] \) is similarly defined as

\[
\begin{align*}
X[\psi/X] &\equiv \psi \\
Y[\psi/X] &\equiv Y \text{ if } Y \neq X \\
x[\psi/X] &\equiv x \\
\bot[\psi/X] &\equiv \bot \\
\sigma[\psi/X] &\equiv \sigma
\end{align*}
\]

B Matching Logic Semantics

Let us fix a matching logic signature \( \Sigma \), which is a set of (unsorted) symbols.

**Definition 10.** A *(matching logic) \( \Sigma \)-model* is a tuple \( (M, \cdot : M \times M \to \mathcal{P}(M), \{\sigma_M\}_{\sigma \in \Sigma}) \), where

- \( M \) is a nonempty *carrier set*,
- \( \cdot \) is an interpretation for application, and
- Each \( \sigma_M \subseteq M \) is an interpretation for the symbol \( \sigma \in \Sigma \).

**Definition 11.** Given a model \( M \), a *valuation* \( \rho \) is a pair of maps \( \rho_1 : EV \to M \) and \( \rho_2 : SV \to \mathcal{P}(M) \). For any \( x \in EV \), \( \rho(x) \) denotes \( \rho_1(x) \) and for any \( X \in SV \), \( \rho(X) \) denotes \( \rho_2(X) \). Furthermore, we define \( \rho[x \mapsto a] := (\rho_1[x \mapsto a], \rho_2) \) and \( \rho[X \mapsto A] := (\rho_1, \rho_2[X \mapsto A]) \) for any \( x \in EV \), \( X \in SV \), \( a \in M \), and \( A \subseteq M \).

**Definition 12.** Given a model \( M \) and a valuation \( \rho \), the *interpretation* of a pattern \( \varphi \), \( \langle \varphi \rangle_{M, \rho} \subseteq M \), is defined inductively as

- \( \langle \bot \rangle_{M, \rho} = \emptyset \)
\[ |x|_{M,\rho} = \{\rho(x)\} \text{ for } x \in EV \]
\[ |X|_{M,\rho} = \rho(X) \text{ for } X \in SV \]
\[ |\sigma|_{M,\rho} = \sigma_M \text{ for } \sigma \in \Sigma \]
\[ |\varphi_1 \varphi_2|_{M,\rho} = \bigcup_{a \in |\varphi_1|_{M,\rho}, b \in |\varphi_2|_{M,\rho}} a \cdot b, \text{ i.e. the pointwise extension of } \cdot \]
\[ |\varphi_1 \rightarrow \varphi_2|_{M,\rho} = M \setminus (|\varphi_1|_{M,\rho} \setminus |\varphi_2|_{M,\rho}) \]
\[ |\exists x. \varphi|_{M,\rho} = \bigcap_{a \in M} |\varphi|_{M,\rho[\rightarrow x = a]} \]
\[ |\mu X. \varphi|_{M,\rho} = \bigcap \{A \subseteq M \mid |\varphi|_{M,\rho[X \mapsto A]} \supseteq A\} \text{, i.e. the least fixpoint of the function } A \mapsto |\varphi|_{M,\rho[\rightarrow X \mapsto A]} \text{ (by the Knaster-Tarski fixpoint theorem [49])} \]

A consequence from Definition 12 is that the intuition about other derived notations such as \( \land \) and \( \lor \) (Section 4.1.2) work as intended:

**Proposition 13.** For any model \( M \), valuation \( \rho \), and patterns \( \varphi \), \( \varphi_1 \), and \( \varphi_2 \), the following holds:
\[
\begin{align*}
|\top|_{M,\rho} &= M \\
|\varphi_1 \lor \varphi_2|_{M,\rho} &= |\varphi_1|_{M,\rho} \cup |\varphi_2|_{M,\rho} \\
|\varphi_1 \land \varphi_2|_{M,\rho} &= |\varphi_1|_{M,\rho} \cap |\varphi_2|_{M,\rho} \\
|\neg \varphi|_{M,\rho} &= M \setminus |\varphi|_{M,\rho} \\
|\forall x. \varphi|_{M,\rho} &= \bigcap_{a \in M} |\varphi|_{M,\rho[\rightarrow x = a]} \\
|\nu x. \varphi|_{M,\rho} &= \bigcup \{A \subseteq M \mid |\varphi|_{M,\rho[X \mapsto A]} \supseteq A\}
\end{align*}
\]

**Proof.** See [8].

We now define the satisfiability relation and semantic entailment.

**Definition 14.** Given a model \( M \), we say that \( M \) satisfies a pattern \( \varphi \), denoted \( M \models \varphi \), if for any valuation \( \rho \), \( |\varphi|_{M,\rho} = M \).

**Definition 15.** Given a theory \( \Gamma \), we say that \( \Gamma \) semantically entails \( \varphi \), denoted \( \Gamma \models \varphi \), if for any \( M \) such that \( M \models \psi \) for all \( \psi \in \Gamma \), we have \( M \models \varphi \).

Connecting with the proof system in Figure 3, we have the following soundness theorem:

**Theorem 16** (Soundness). For any theory \( \Gamma \) and pattern \( \varphi \), if \( \Gamma \vdash \varphi \) then \( \Gamma \models \varphi \).

**Proof.** See [8].

C A 240-line Formalization of Matching Logic in Metamath

Below is the matching logic formalization we use in our work. For the usage of Metamath, see [36] or a shorter introduction in [5].

\[
\begin{verbatim}
$ ( Matching Logic Proof Checker in 240 LOC $)  
$C #Pattern #ElementVariable #SetVariable #Variable #Symbol $.  
$C #Positive #Negative #Fresh #ApplicationContext #Substitution #Notation |- $.  
$C \bot \imp \app \exists \mu ( ) $.  
$y ph0 ph1 ph2 ph3 ph4 ph5 x y X Y xx yY sg0 $.
\end{verbatim}

31
$( Syntax )$

ph0-is-pattern $f$ #Pattern ph0 $.
ph1-is-pattern $f$ #Pattern ph1 $.
ph2-is-pattern $f$ #Pattern ph2 $.
ph3-is-pattern $f$ #Pattern ph3 $.
ph4-is-pattern $f$ #Pattern ph4 $.
ph5-is-pattern $f$ #Pattern ph5 $.
x-is-element-var $f$ #ElementVariable x $.
y-is-element-var $f$ #ElementVariable y $.
xX-is-var $f$ #Variable xX $.
yY-is-element-var $f$ #Variable yY $.
sg0-is-symbol $f$ #Symbol sg0 $.

$( Positive occurrence )$

element-var-is-var $a$ #Variable x $.
set-var-is-var $a$ #Variable X $.
var-is-pattern $a$ #Pattern xX $.
symbol-is-pattern $a$ #Pattern sg0 $.

bot-is-pattern $a$ #Pattern $\bot$. 
imp-is-pattern $a$ #Pattern ( \imp ph0 ph1 ) $.
app-is-pattern $a$ #Pattern ( \app ph0 ph1 ) $.
exists-is-pattern $a$ #Pattern ( \exists x ph0 ) $.

${\{ \mu-is-pattern.0 $e$ #Positive X ph0 $.

mu-is-pattern $a$ #Pattern ( \mu X ph0 ) $.$} 

$( Negative occurrence )$

negative-in-symbol $a$ #Negative xX sg0 $.
negative-in-bot $a$ #Negative xX $\bot$. 

${\{ \neg-x-x yY $.

negative-in-var $a$ #Negative xX yY $.$} 

${\{ \neg-x-x ph0 $.$}

negative-in-imp $a$ #Negative xX ph0 $.
negative-in-imp.1 $e$ #Negative xX ph1 $.

${\{ \neg-x-x ( \imp ph0 ph1 ) $.$} 

negative-in-app $a$ #Negative xX $.

negative-in-app.0 $e$ #Negative xX ph0 $.$

negative-in-app.1 $e$ #Negative xX ph1 $.$

${\{ \neg-x-x ( \app ph0 ph1 ) $.$} 

negative-in-exists $a$ #Negative xX $.$.

negative-in-mu.0 $e$ #Negative xX $.$.

${\{ \neg-x-x ( \mu X ph0 ) $.$} 

$d x x ph0 $.

negative-disjoint $a$ #Negative xX ph0 $.$} 

$( Negative occurrence )$

negative-in-symbol $a$ #Negative xX sg0 $.
negative-in-bot $a$ #Negative xX $\bot$. 

${\{ \neg-d x x yY $.

negative-in-var $a$ #Negative xX yY $.$} 

${\{ \neg-d-x-x ph0 $.$}

negative-in-imp $a$ #Negative xX ph0 $.
negative-in-imp.1 $e$ #Negative xX ph1 $.

${\{ \neg-d-x-x ( \imp ph0 ph1 ) $.$} 

negative-in-app $a$ #Negative xX $.

negative-in-app.0 $e$ #Negative xX ph0 $.$

negative-in-app.1 $e$ #Negative xX ph1 $.$

${\{ \neg-d-x-x ( \app ph0 ph1 ) $.$} 

negative-in-exists $a$ #Negative xX $.$.

negative-in-mu.0 $e$ #Negative xX $.$.

${\{ \neg-d-x-x ( \mu X ph0 ) $.$} 

$d x x ph0 $.

negative-disjoint $a$ #Negative xX ph0 $.$} 

32
( Free variable $)

fresh-in-symbol  $a \#Fresh xX \text{ sg0 }$.
fresh-in-bot      $a \#Fresh xX \text{ \bot }$.
fresh-in-exists-shadowed $a \#Fresh x ( \{x \mid exists x \text{ ph0} \} )$.
fresh-in-mu-shadowed  $a \#Fresh X ( \mu X \text{ ph0} )$.

$\{ \begin{array}{l} 
$d xX \text{ ph0 }$.
fresh-disjoint $a \#Fresh xX \text{ ph0 }$.
\end{array} \}$

fresh-in-imp.0 $e \#Fresh xX \text{ ph0 }$.
fresh-in-imp.1 $e \#Fresh xX \text{ ph1 }$.

$\{ \begin{array}{l} 
$ fresh-in-app.0 $e \#Fresh xX \text{ ph0 }$.
$ fresh-in-app.1 $e \#Fresh xX \text{ ph1 }$.
\end{array} \}$

$\{ \begin{array}{l} 
新鲜性 xX \text{ ph0 }$.
$ fresh-in-exists.0 $e \#Fresh xX \{ \{x \mid exists x \text{ ph0} \} \}.$
$ fresh-in-app.0 $e \#Fresh xX \{ \{ \mu X \text{ ph0} \} \}.$
\end{array} \}$

$\{ \begin{array}{l} 
$ fresh-in-exists-shadowed $a \#Fresh x ( \exists x \text{ ph0} )$.
$ fresh-in-mu-shadowed $a \#Fresh X ( \mu X \text{ ph0} )$.
\end{array} \}$

$\{ \begin{array}{l} 
$ substitution-var-same $a \#Substitution ph0 xX ph0 xX$.
$ substitution-symbol $a \#Substitution sg0 sg0 ph0 xX$.
$ substitution-bot $a \#Substitution \text{ \bot } \text{ bot } \text{ ph0 } xX$.
$ substitution-identity $a \#Substitution ph0 ph0 xX xX$.
\end{array} \}$

$\{ \begin{array}{l} 
$ substitution-exists-shadowed $a \#Substitution ( \exists x \text{ ph1} ) ( \{ \exists x \text{ ph1} \} \text{ ph0} xX$.
substitution-mu-shadowed  $a \#Substitution ( \mu X \text{ ph1} ) ( \mu X \text{ ph1} ) \text{ ph0 X}$.
\end{array} \}$

$\{ \begin{array}{l} 
$ substitution-imp.0 $e \#Substitution ph1 ph3 ph0 X$.
$ substitution-imp.1 $e \#Substitution ph2 ph4 ph0 xX$.
\end{array} \}$

$\{ \begin{array}{l} 
$ substitution-app.0 $e \#Substitution ph1 ph3 ph0 xX$.
$ substitution-app.1 $e \#Substitution ph2 ph4 ph0 xX$.
\end{array} \}$

$\{ \begin{array}{l} 
$ substitution-exists.0 $e \#Substitution ph2 ph1 y x$.
$ substitution-exists.1 $e \#Substitution ph3 ph2 ph0 xX$.
\end{array} \}$

$\{ \begin{array}{l} 
$ substitution-exists-shadowed $a \#Substitution ( \exists y \text{ ph3} ) ( \{ \exists x \text{ ph1} \} ) \text{ ph0 xX}$.
\end{array} \}$

$\{ \begin{array}{l} 
$ substitution-mu.0 $e \#Substitution ph2 ph1 Y$.
$ substitution-mu.1 $e \#Substitution ph3 ph2 ph0 xX$.
\end{array} \}$

$\{ \begin{array}{l} 
$ yX-free-in-ph0 $e \#Fresh yX \text{ ph0}$.
\end{array} \}$

$\{ \begin{array}{l} 
$ substitution-fold.0 $e \#Substitution ph1 ph3 yX$.
$ substitution-fold $a \#Substitution ph2 ph0 xX$.
\end{array} \}$

$\{ \begin{array}{l} 
$ substitution-unfold.0 $e \#Substitution ph2 ph0 xX$.
$ substitution-unfold $a \#Substitution ph2 ph1 ph3 yX$.
\end{array} \}$

$\{ \begin{array}{l} 
$ substitution-inverse.0 $e \#Fresh xX \text{ ph0}$.
$ substitution-inverse.1 $e \#Substitution ph1 ph0 xX$.
\end{array} \}$

$\{ \begin{array}{l} 
$ substitution-fresh.0 $e \#Fresh xX \text{ ph0}$.
$ substitution-fresh $a \#Substitution ph0 ph0 xX$.
\end{array} \}$

$\{ \begin{array}{l} 
$ application-context-var $a \#ApplicationContext xX xX$.
\end{array} \}$
application-context-app-left.0 $e \#ApplicationContext xX ph0 \$.
application-context-app-left $a \#ApplicationContext xX ( \app ph0 ph1 ) \$. \}$
$( xX ph0 \$.
application-context-app-right.0 $e \#ApplicationContext xX ph1 \$.
application-context-app-right $a \#ApplicationContext xX ( \app ph0 ph1 ) \$. \}$

$( Notation )
notation-reflexivity $a \#Notation ph0 ph0 \$.
notation-symmetry.0 $e \#Notation ph0 ph1 \$.
notation-symmetry $a \#Notation ph1 ph0 \$. \}$
notation-transitivity.0 $e \#Notation ph0 ph1 \$.
notation-transitivity.1 $e \#Notation ph1 ph2 \$.
notation-transitivity $a \#Notation ph0 ph2 \$. \}$
notation-positive.0 $e \#Positive xX ph0 \$.
notation-positive.1 $e \#Notation ph1 ph0 \$.
notation-positive $a \#Positive xX ph1 \$. \}$
notation-negative.0 $e \#Negative xX ph0 \$.
notation-negative.1 $e \#Notation ph1 ph0 \$.
notation-negative $a \#Negative xX ph1 \$. \}$
notation-fresh.0 $e \#Fresh xX ph0 \$.
notation-fresh.1 $e \#Notation ph1 ph0 \$.
notation-fresh $a \#Fresh xX ph1 \$. \}$
notation-application-context.0 $e \#ApplicationContext xX ph0 \$.
notation-application-context.1 $e \#Notation ph1 ph0 \$.
notation-application-context $a \#ApplicationContext xX ph1 \$. \}$
notation-proof.0 $e |- ph0 \$.
notation-proof.1 $e |- ph1 \$. \}$
notation-proof $a |- ph1 \$. \}$
notation-imp.0 $e \#Notation ph0 ph2 \$.
notation-imp.1 $e \#Notation ph1 ph3 \$.
notation-imp $a \#Notation ( \imp ph0 ph1 ) ( \imp ph2 ph3 ) \$. \}$
notation-substitution.0 $e \#Substitution ph0 ph1 ph2 xX \$.
notation-substitution.1 $e \#Notation ph3 ph0 \$. \}$
notation-substitution $a \#Notation ph4 ph1 \$. \}$
notation-substitution.2 $e \#Notation ph4 ph1 \$.
notation-substitution.3 $e \#Notation ph5 ph2 \$.
notation-substitution $a \#Substitution ph3 ph4 ph5 xX \$. \}$
notation-application-context.0 $e \#ApplicationContext xX ph0 \$.
notation-application-context.1 $e \#Notation ph1 ph0 \$.
notation-application-context $a \#ApplicationContext xX ph1 \$. \}$
notation-proof.0 $e |- ph0 \$.
notation-proof.1 $e |- ph1 \$. \}$
notation-proof $a |- ph1 \$. \}$
notation-imp.0 $e \#Notation ph0 ph2 \$.
notation-imp.1 $e \#Notation ph1 ph3 \$.
notation-imp $a \#Notation ( \imp ph0 ph1 ) ( \imp ph2 ph3 ) \$. \}$
notation-app.0 $e \#Notation ph0 ph2 \$.
notation-app.1 $e \#Notation ph1 ph3 \$.
notation-app $a \#Notation ( \app ph0 ph1 ) ( \app ph2 ph3 ) \$. \}$
notation-exists.0 $e \#Notation ph0 ph1 \$.
notation-exists $a \#Notation ( \exists x ph0 ) ( \exists x ph1 ) \$. \}$
notation-mu.0 $e \#Notation ( \mu X ph0 ) ( \mu X ph1 ) \$. \}$

$( Defining not, or, and )
\not \or \and 
not-is-pattern $a \#Pattern ( \not ph0 ) \$. \}$
or-is-pattern $a \#Pattern ( \or ph0 ph1 ) \$. \}$
and-is-pattern $a \#Pattern ( \and ph0 ph1 ) \$. \}$
not-is-sugar $a \#Notation ( \not ph0 ) ( \imp ph0 \bot ) \$. \}$
or-is-sugar $a \#Notation ( \or ph0 ph1 ) ( \imp ( \not ph0 ) ph1 ) \$. \}$
and-is-sugar $a \#Notation ( \and ph0 ph1 ) ( \not ( \or ( \not ph0 ) ( \not ph1 ) ) ) \$. \}$

$( Proof system )
proof-rule-prop.1 $a |- ( \imp ph0 ( \imp ph1 ph0 ) ) \$. \}$
proof-rule-prop.2 $a |- ( \imp ( \imp ph0 ( \imp ph1 ph2 ) ) ( \imp ( \imp ph0 ph1 ) ( \imp ph0 ph2 ) ) ) \$. \}$
proof-rule-prop.3 $a |- ( \imp ( \imp ( \imp ph0 \bot ) \bot ) ph0 ) \$. \}$
proof-rule-prop.0 $e |- ( \imp ph0 ph1 ) \$. \}$
proof-rule-prop.1 $a |- ph0 \$. \}$
proof-rule-prop $a |- ph1 \$. \}$

34
D Benchmarks used in Evaluation

We present the language definitions of REG (Appendix D.1) and PCF (Appendix D.2) in this appendix. The definition of IMP can be found in the main text in Figure 2.

The functional specifications of SUM in three languages are shown in Appendix D.3, Appendix D.4, and Appendix D.5. We refer reader to our anonymous repository [1] for the specifications of EXP and COLLATZ.

D.1 Syntax and Semantics of REG in \( \mathbb{K} \)
module REG-SYNTAX
imports DOMAINS-SYNTAX
syntax ControlCommand ::= "exec" | ret(Int)
syntax Immediate ::= Int
syntax Register ::= "r0" | "r1" | "r2" | "r3"
syntax Operand ::= Immediate | Register
syntax Address ::= rel(Int) | abs(Int)
syntax Instruction ::= 
  | "load" Register "," Operand [strict(2)]
  | "store" Operand "," Operand [seqstrict]
  | "add" Register "," Operand "," Operand [strict(2, 3)]
  | "sub" Register "," Operand "," Operand [strict(2, 3)]
  | "mul" Register "," Operand "," Operand [strict(2, 3)]
  | "div" Register "," Operand "," Operand [strict(2, 3)]
  | "le" Register "," Operand "," Operand [strict(2, 3)]
  | "not" Register "," Operand [strict(2)]
  | "br" Operand "," Address "," Address [strict(1)]
  | "jump" Address
  | "ret" Operand [strict]
endmodule

module REG // REG semantics
imports DOMAINS
imports REG-SYNTAX
configuration
  <T> <k> exec </k>
  <pc> 0 </pc>
  <r0> 0 </r0>
  <r1> 0 </r1>
  <r2> 0 </r2>
  <r3> 0 </r3>
  <imem> .Map </imem>
  <dmem> .Map </dmem> </T>

syntax KResult ::= Int

// Read and execute the next instruction
rule <k> exec == I ~> exec ... </k>
  <pc> A +Int 1 </pc>
  <imem> A |-> I ... </imem>

// Register lookup
rule <k> r0 == V ... </k> <r0> V </r0>
rule <k> r1 == V ... </k> <r1> V </r1>
rule <k> r2 == V ... </k> <r2> V </r2>
rule <k> r3 == V ... </k> <r3> V </r3>

// Register set
syntax KItem ::= setRegister(Register, KItem) [strict(2)]
rule <k> setRegister(r0, V:Int) == . ... </k> <r0> _ => V </r0>
rule <k> setRegister(r1, V:Int) == . ... </k> <r1> _ => V </r1>
rule <k> setRegister(r2, V:Int) == . ... </k> <r2> _ => V </r2>
rule <k> setRegister(r3, V:Int) == . ... </k> <r3> _ => V </r3>

rule <k> ret V:Int ~> exec == ret(V) </k>
  <pc> _ => 0 </pc>
  <r0> _ => 0 </r0>
  <r1> _ => 0 </r1>
  <r2> _ => 0 </r2>
  <r3> _ => 0 </r3>
  <dmem> _ => .Map </dmem>

rule move R:Register, V:Int == setRegister(R, V)
D.2 Syntax and Semantics of PCF in $\mathbb{K}$

```k
module PCF-SYNTAX
  imports DOMAINS-SYNTAX
  syntax Type ::= "int" | "bool" | Type "->" Type
  syntax Value ::= Int | Bool | lam(Id, Type, Term)
  syntax Term ::= Id | if(Term, Term, Term) [strict(1)]
                 | add(Term, Term)
                 | mul(Term, Term) [seqstrict]
                 | div(Term, Term) [seqstrict]
                 | le(Term, Term) [seqstrict]
                 | not(Term) [strict]
                 | app(Term, Term) [seqstrict]
                 | fix(Id, Type, Term)
endmodule

module PCF // PCF semantics
  imports DOMAINS
  imports PCF-SYNTAX
  syntax KResult ::= Value
  // Substitution that COULD potentially have variable capturing
  // but since we are dealing with closed terms this is fine
  syntax Term ::= substitute(Term, Id, Term) [function, functional]
  rule substitute(V:Int, _, _) => V
  rule substitute(B:Bool, _, _) => B
  rule substitute(X:Id, X, T) => T
  rule substitute(Y:Id, X, _) => Y requires X /= K Y
  rule substitute(if(T1, T2, T3), X, T) => if(substitute(T1, X, T),
                                           substitute(T2, X, T), substitute(T3, X, T))
  rule substitute(add(T1, T2), X, T) => add(substitute(T1, X, T), substitute(T2, X, T))
  rule substitute(mul(T1, T2), X, T) => mul(substitute(T1, X, T), substitute(T2, X, T))
  rule substitute(div(T1, T2), X, T) => div(substitute(T1, X, T), substitute(T2, X, T))
  rule substitute(le(T1, T2), X, T) => le(substitute(T1, X, T), substitute(T2, X, T))
  rule substitute(not(T'), X, T) => not(substitute(T', X, T))
  rule substitute(app(T1, T2), X, T) => app(substitute(T1, X, T), substitute(T2, X, T))
  rule substitute(lam(X, P, T'), X, _) => lam(X, P, T')
endmodule
```
D.3 Specification of sum in IMP

```plaintext
module IMP-SUM-SPEC
  imports IMP

syntax Id ::= "n" [token] | "sum" [token]

claim <k> int n, sum;
  n = N:Int; sum = 0;
  while (!(n <= 0)) { sum = sum + n; n = n + -1; }
=> .
</k>

<state> .Map => n |-> 0 sum |-> ((N +Int 1) *Int N /Int 2) </state>
requires N >=Int 0

// Loop invariant
claim <k> while (!(n = 0)) { sum = sum + n; n = n + -1; } => . . . </k>

<state>
  n |-> (N:Int = 0)
  sum |-> (S:Int = S +Int ((N +Int 1) *Int N /Int 2))
</state>
requires N >=Int 0
endmodule
```

D.4 Specification of sum in REG

```plaintext
module REG-SUM-SPEC
  imports REG

claim <k> exec => ret(((N +Int 1) *Int N) /Int 2) </k>

<pc> 0 </pc>
<reg> N => 0 </reg>
<r1> 0 </r1>
```
D.5 Specification of \texttt{SUM} in PCF

\begin{verbatim}
module PCF-SUM-SPEC
imports PCF

syntax Id ::= "f" [token] | "x" [token]

claim add(0,
  app(
    fix(f, int -> int, lam(x, int, if(le(x, 0), 0, add(x, app(f, add(x, -1)))))),
    N:Int
  )
) => (((N +Int 1) *Int N) /Int 2)
requires N =>Int 0

endmodule
\end{verbatim}
E A Wrong Reachability Proof in [14]

In [14, Fig. 6(c)], an example of a reachability proof for \( \text{SUM} \) is presented, but it contains an invalid application of a proof rule, which we will discuss below. This shows how error-prone a reachability proof can be even in a small example, which necessitates the use of machine-checkable proofs in our approach.

Recall the following \( \text{SUM} \) program, which computes the sum \( 1 + \cdots + n \):

```c
int n, s; n = n; s = 0;
while (!(!n <= 0)) { s = s + n; n = n - 1; }
```

Here \( n \) is a free variable (of integer sort) at the matching logic level, therefore it is implicitly universally quantified.

When loaded into \( K \) alongside with the state cell, the program is encoded as the configuration

\[
\langle \langle \text{SUM} \rangle \rangle_k \langle \cdot \rangle_{\text{Map}} \langle \text{state} \rangle_T
\]

(24)

Then we can state the functional correctness of \( \text{SUM} \) as the following reachability judgment

\[
A \vdash \text{reach} \emptyset \langle \langle \text{SUM} \rangle \rangle_k \langle \cdot \rangle_{\text{Map}} \langle \text{state} \rangle_T \Rightarrow \langle \langle \text{LOOP} \rangle \rangle_k \langle \cdot \rangle_{\text{Map}} \langle \text{state} \rangle_T
\]

where \( A \) contains all rewrite rules in the semantics of IMP (Figure 2). By axioms in \( A \), we can see that (after multiple applications of (Axiom) and (Transitivity)):

\[
A \vdash \text{reach} \emptyset \langle \langle \text{SUM} \rangle \rangle_k \langle \cdot \rangle_{\text{Map}} \langle \text{state} \rangle_T \Rightarrow \langle \langle \text{LOOP} \rangle \rangle_k \langle \cdot \rangle_{\text{Map}} \langle \text{state} \rangle_T
\]

(25)

Now, we can reduce this goal to the following more general claim (metavariabes \( n \) and \( m \) denote distinct matching logic variables):

\[
A \vdash \text{reach} \emptyset (\exists m. \langle \langle \text{LOOP} \rangle \rangle_k \langle \cdot \rangle_{\text{Map}} \langle \text{state} \rangle_T \Rightarrow \langle \langle \text{LOOP} \rangle \rangle_k \langle \cdot \rangle_{\text{Map}} \langle \text{state} \rangle_T)
\]

(26)

Because we have the subsumption:

\[
\mathcal{T} \models \langle \langle \text{LOOP} \rangle \rangle_k \langle \cdot \rangle_{\text{Map}} \langle \text{state} \rangle_T \Rightarrow \exists m. \langle \langle \text{LOOP} \rangle \rangle_k \langle \cdot \rangle_{\text{Map}} \langle \text{state} \rangle_T
\]

The generalized claim (26) is exactly our loop invariant. It says that for any integer \( m \), the configuration on the left-hand side of (26) reaches the right-hand side. For simplicity, let

\[
\varphi_{\text{inv}}(m) \equiv \langle \langle \text{LOOP} \rangle \rangle_k \langle \cdot \rangle_{\text{Map}} \langle \text{state} \rangle_T
\]

\[
\varphi_{\text{post}} \equiv \langle \langle \text{LOOP} \rangle \rangle_k \langle \cdot \rangle_{\text{Map}} \langle \text{state} \rangle_T
\]

The proof in [14, Fig. 6(c)] up to this point is correct. However, the next steps applied in the proof are the following

\[
\begin{array}{c}
\vdots \\
\text{(Circularity)} \\
\text{(Abstraction)} \\
A \vdash \text{reach} \varphi_{\text{inv}}(m) \Rightarrow \varphi_{\text{post}} \\
A \vdash \text{reach} \varphi_{\text{inv}}(m) \Rightarrow \varphi_{\text{post}} \\
(26) = A \vdash \text{reach} (\exists m. \varphi_{\text{inv}}(m)) \Rightarrow \varphi_{\text{post}}
\end{array}
\]

40
This is problematic because the left-hand side is transformed from $\exists m. \varphi_{\text{inv}}(m)$ to $\varphi_{\text{inv}}(m)$, which makes the variable $m$ unable to be instantiated by another term later when the invariant is used. So instead, the proof should switch the order of (Abstraction) and (Circularity) so that we can keep the stronger version in the circularity set:

$$
\begin{align*}
\vdash A \Gamma_{\text{reach}} \{ (\exists m. \varphi_{\text{inv}}(m)) \Rightarrow \varphi_{\text{post}} \} & \varphi_{\text{inv}}(m) \Rightarrow \varphi_{\text{post}}, \\
\vdash A \Gamma_{\text{reach}} \{ (\exists m. \varphi_{\text{inv}}(m)) \Rightarrow \varphi_{\text{post}} \} & (\exists m. \varphi_{\text{inv}}(m)) \Rightarrow \varphi_{\text{post}} \\
= & A \Gamma_{\text{reach}} (\exists m. \varphi_{\text{inv}}(m)) \Rightarrow \varphi_{\text{post}}
\end{align*}
$$

F  Set Circularity

As discussed in Section 9.2.2 the reachability prover in $K$ (Algorithm 1) uses the set circularity rule at the beginning to allow inductive application of all the claims:

$$
\begin{align*}
\vdash A \Gamma_{\text{reach}} \varphi \Rightarrow \psi & \text{ for all } (\varphi \Rightarrow \psi) \in I, \\
\vdash A \Gamma_{\text{reach}} \varphi \Rightarrow \psi & \text{ for all } (\varphi \Rightarrow \psi) \in I
\end{align*}
$$

[42, Lemma 5] proves this rule by apply a analogy of cut lemma in reachability logic inductively on the proof of $A \vdash_{\text{reach}} \varphi \Rightarrow \psi$. We can reproduce this proof during proof generation at the matching logic level. Suppose for simplicity that we have two inter-dependent claims $S = \{ \varphi_1 \Rightarrow \psi_1, \varphi_2 \Rightarrow \psi_2 \}$ such that a proof of $A \vdash_{\text{reach}} \varphi_1 \Rightarrow \psi_1$ need to invoke $\varphi_2 \Rightarrow \psi_2$, and vice versa. Therefore, the application of (Set Circularity) cannot be directly reduced to (Circularity).

Without loss of generality, suppose we first generate a proof of

$$
A \vdash_{\text{reach}} (\varphi_2 \Rightarrow \psi_2, \varphi_1 \Rightarrow \psi_1) \varphi_1 \Rightarrow \psi_1
$$

which can be strengthened by (Circularity) to

$$
A \vdash_{\text{reach}} (\varphi_2 \Rightarrow \psi_2) \varphi_1 \Rightarrow \psi_1
$$

Then when we generate a proof for $\varphi_2 \Rightarrow \psi_2$, whenever we need to use $\varphi_1 \Rightarrow \psi_1$, we can apply (27). Thus, we can get a proof of

$$
A \vdash_{\text{reach}} \varphi_2 \Rightarrow^+ \psi_2
$$

We assume $\Rightarrow^+$ here because otherwise $\varphi_2 \rightarrow \psi_2$, in which case the subsumption can be directly proved in the proof of $\varphi_1 \Rightarrow \psi_1$.

In matching logic encoding (see Section 5.2), (27) and (28) are

$$
\begin{align*}
\vdash \forall \Box A & \land \forall \Box (\varphi_2 \Rightarrow^+ \psi_2) \rightarrow (\varphi_1 \Rightarrow_{\text{reach}} \psi_1) \\
\vdash \forall \Box A & \rightarrow (\varphi_2 \Rightarrow^+ \psi_2)
\end{align*}
$$

where $\forall \Box A$ abbreviates for $\bigwedge_{\varphi \Rightarrow \psi} \forall \Box (\forall \Box \varphi (\varphi) \cup \forall \Box \psi (\Box (\varphi \Rightarrow^+ \psi)))$ (see Section 5.2).

Since (30) can be weakened and quantified to

$$
\vdash \forall \Box A \rightarrow \forall \Box (\varphi_2 \Rightarrow^+ \psi_2)
$$

By propositional reasoning, we can strengthen the first claim (29) to

$$
\vdash \forall \Box A \rightarrow (\varphi_1 \Rightarrow_{\text{reach}} \psi_1)
$$

The last step is essentially a propositional version of cut. So in general, given a set of claims $S$, we can prove it in any order. In each step, the premises of the proven claims should only mention the unproven ones, because the other dependencies can be eliminated through (Circularity) and cut as the example above. This invariant ensures that we would have eliminated all circularity premises in the end.

41
Figure 6: Statistics on compressed proofs. The columns are proof object names, proof size before/after compression, proof checking time using \textit{mmverify} before/after using incremental decompression. Note that the proof checking time shown here should not be compared to that in Figure 5, because they use different Metamath verifiers. Here, we are using the online Metamath verifier \textit{mmverify}, which runs more slowly than the official Metamath verifier (see text in Appendix G for details).

G Proof Compression

We experimented with using a generic lossless compression tool \textit{xz} \cite{XZ} to incrementally compress and decompress proof objects. We show in Figure 6 some statistics.

Note that in Figure 6 we are using a different online proof checker (\textit{mmverify} \cite{Metamath}) rather than the official Metamath implementation used in Section 8, because the latter only supports reading a complete proof object and does not allow incremental proof checking. Since \textit{mmverify} is implemented in Python, it is slower than the official C implementation.

The statistics in Figure 6 show that:

- General compression can greatly reduce the proof object size, and the compression ratio is in the order of 100 : 1.
- An incremental decompression algorithm combined with an online proof checker is as fast as in the case without compression.

Therefore, we conclude that the space-efficiency of our proof object has a lot of room for improvement and the current large sizes are primarily due to an inefficient encoding rather than the amount of information contained in the proof.

H Metamath Encoding of (Transitivity) and (Circularity)

Section 5.2 presented some examples of reachability lemmas we have in our matching logic database. In this appendix, we elaborate more on their Metamath encoding. In our Metamath database, (Transitivity) is stated as

\begin{verbatim}
$(
  tr.0 s |- ( \in-sort ph1 s ) $. $( \text{ sorting hypotheses } )$
  tr.1 s |- ( \in-sort ph2 s ) $. 
  tr.2 s |- ( \in-sort ph3 s ) $. 
  tr.3 s |- ( \in-sort ph4 s ) $. 
  tr.4 s |- ( \valid s ( \implies s ( \circularity s ph1 ) ( \reaches-plus s ph2 ph3 ) ) ) $. 
  tr.5 s |- ( \valid s ( \implies s ( \always s ph1 ) ( \reaches-star s ph3 ph4 ) ) ) $. 
  tr s |- ( \valid s ( \implies s ( \circularity s ph1 ) ( \reaches-plus s ph2 ph4 ) ) ) 
  $= $( proof omitted ) $. 
)\end{verbatim}
where \(\text{valid s}, \text{implies s}, \text{circularity s}, \text{always s}, \text{reaches-plus s}\) are sorted versions of validity, implication, circularity (\(\circ\square\)), always (\(\square\)), and (one-path) reachability (\(\Rightarrow_{\text{reach}}^+\)), respectively.

For (Circularity), since the operation of quantifying over all free variables (\(\forall \varphi\)) cannot be directly encoded in Metamath, it is broken into two cases:

\[
\begin{align*}
&\text{cc-1.0 } \vdash (\text{\in-sort ph1 s}) \\
&\text{cc-1.1 } \vdash (\text{valid s (implies s (circularity s ph1) ph1)}) \\
&\text{cc-1 } \vdash (\text{valid s (or s (non-well-founded s) ph1)}) \\
&\quad \vdash (\text{proof omitted}) \\
&\text{cc-2.0 } \vdash (\text{\in-sort ph1 s}) \\
&\text{cc-2.1 } \vdash (\text{\in-sort ph2 s}) \\
&\text{cc-2 } \vdash (\text{eq s (or s (non-well-founded s) (reaches-plus s ph1 ph2))})
\end{align*}
\]

where \(\text{eq}\) denotes equality (Section 4.3), and (\(\text{non-well-founded s}\)) denotes \(\neg \mu X. \circ X\), whose semantics is the set of elements that have an infinite trace in the transition system.

To invoke (Circularity), starting from the original hypotheses \(\Gamma \vdash \circ \square (\forall \varphi \Rightarrow_{\text{reach}} \psi) \rightarrow \varphi \Rightarrow_{\text{reach}} \psi\), we first dynamically generate a proof to quantify all free variables on the right hand side:

\[
\begin{align*}
\Gamma &\vdash \circ \square (\forall \varphi \Rightarrow_{\text{reach}} \psi) \rightarrow (\forall \varphi \Rightarrow_{\text{reach}} \psi)
\end{align*}
\]

Then cc-1 is applied to get

\[
\begin{align*}
\Gamma &\vdash \neg \mu X. \circ X \lor (\forall \varphi \Rightarrow_{\text{reach}} \psi)
\end{align*}
\]

which by cc-2, implies \(\Gamma \vdash \forall \varphi \Rightarrow_{\text{reach}} \psi\). Finally, the quantifiers are eliminated to get the desired \(\Gamma \vdash \varphi \Rightarrow_{\text{reach}} \psi\).