Towards a Trustworthy Semantics-Based Language Framework via Proof Generation

Anonymous Authors

Abstract. We pursue the vision of an *ideal language framework*, where programming language designers only need to define the formal *syntax* and *semantics* of their languages, and all language tools are automatically generated by the framework. Due to the complexity of such a language framework, it is a big challenge to ensure its trustworthiness and to establish the correctness of the autogenerated language tools. In this paper, we propose an innovative approach based on *proof generation*. The key idea is to generate proof objects as correctness certificates for each individual task that the language tools conduct, on a case-by-case basis, and use a trustworthy proof checker to check the proof objects. This way, we avoid formally verifying the entire framework, which is practically impossible, and thus can make the language framework both *practical* and *trustworthy*. As a first step, we formalize program execution as mathematical proofs and generate their complete proof objects. The experimental result shows that the performance of our proof object generation and proof checking is very promising.

1 Introduction

Unlike natural languages that allow vagueness and ambiguity, programming languages must be precise and unambiguous. Only with rigorous definitions of programming languages, called the *formal semantics*, can we guarantee the reliability, safety, and security of computing systems.

Our vision is thus an *ideal language framework* based on the formal semantics of programming languages. 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, and all language tools are automatically generated by the framework. The *correctness* of these language tools is established by generating complete mathematical proofs as certificates that can be automatically machine-checked by a trustworthy proof checker.

The $\mathbb{K}$ language framework (`http:kframework.org`) is in pursuit of the above ideal vision. It provides a simple and intuitive front end language (i.e., a meta-language) for language designers to define the formal syntax and semantics of other programming languages. From such a formal language definition, the framework automatically generates a set of language tools, including a parser, an interpreter, a deductive verifier, a program equivalence checker, among many others [9,25]. $\mathbb{K}$ has obtained much success in practice, and has been used to define the complete executable formal semantics of many real-world languages, such as C [12], Java [3], JavaScript [21], Python [13], Ethereum virtual machines

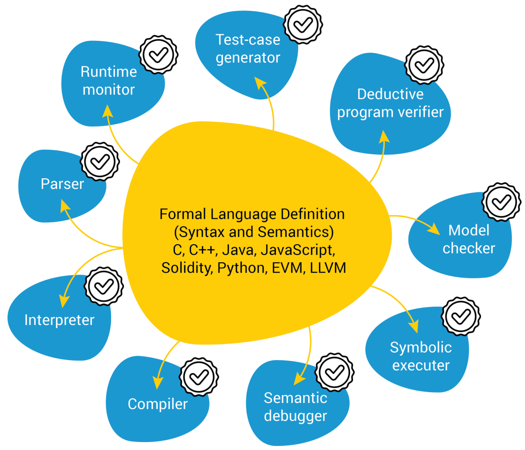

Fig. 1: An ideal language framework vision; language tools are autogenerated, with machine-checkable mathematical proofs as correctness certificates.

byte code [15], and x86-64 [10], from which their implementations and formal analysis tools are automatically generated. Some commercial products [14,17] are powered by these autogenerated implementations and/or tools.

What is *missing* in $\mathbb{K}$ (compared to the ideal vision in Figure 1) is its ability to generate proof objects as correctness certificates. The current $\mathbb{K}$ implementation is a complex artifact with over 500,000 lines of code written in 4 programming languages, with new code committed on a weekly basis. Its code base includes complex data structures, algorithms, optimizations, and heuristics to support the various features such as defining formal language syntax using BNF grammar, defining computation configurations as constructor terms, defining formal semantics using rewrite rules, specifying arbitrary evaluation strategies, and defining the binding behaviors of binders (Section 3). The large code base and rich features make it challenging to formally verify the correctness of $\mathbb{K}$.

Our **main contribution** is the proposal of a *practical approach* to establishing the correctness of a complex language framework, such as $\mathbb{K}$, via *proof object generation*. Our approach consists of the following main components:

1. A small *logical foundation* of $\mathbb{K}$;
2. *Proof parameters* that are provided by $\mathbb{K}$ as the hints for proof generation;
3. A *proof object generator* that generates *proof objects* from proof parameters;
4. A fast and trustworthy third-party *proof checker* that verifies proof objects.

The key idea that makes our approach practical is that we establish the correctness not for the entire framework, but for each individual language tasks that it conducts, on a case-by-case basis. This idea is not limited to $\mathbb{K}$ but also applicable to the existing language frameworks and/or formal semantics approaches.

As a first step, we formalize *program execution* as mathematical proofs and generate their complete proof objects. The experimental result (Table 1) shows promising performance of the proof object generation and proof checking. For example, for a 100-step program execution trace, its complete proof object has 1.6 million lines of code that takes only 5.6 seconds to proof-check.

We organize the rest of the paper as follows. We give an overview of our approach in Section 2. We introduce $\mathbb{K}$ and discuss the generation of proof parameters in Section 3. We discuss *matching logic*—the logical foundation of $\mathbb{K}$—in Section 4. We then compile $\mathbb{K}$ to matching logic in Section 5, and discuss *proof object generation* in Section 6. We discuss the limitations of our current implementation and show the experiment results in Sections 7 and 8, respectively. Finally, we discuss related work and conclude the paper in Section 9.

2 Our Approach Overview

We give an overview of our approach via the following four main components: (1) a logical foundation of $\mathbb{K}$, (2) proof parameters, (3) proof object generation, and (4) a trustworthy proof checker.

Logical Foundation of $\mathbb{K}$. Our approach is based on *matching logic* [23,5]. Matching logic is the *logical foundation* of $\mathbb{K}$, in the following sense:

1. The $\mathbb{K}$ definition (i.e., the language definition in Figure 1) of a programming language L corresponds to a *matching logic theory* Γ^L, which, roughly speaking, consists of a set of logical symbols that represents the formal syntax of L, and a set of logical axioms that specify the formal semantics.
2. All language tools in Figure 1 and all language tasks that $\mathbb{K}$ conducts are formally specified by matching logic formulas. For example, *program execution* is specified (in our approach) by the following matching logic formula:

$$\varphi_{\mathit{init}} \Rightarrow \varphi_{\mathit{final}} \tag{1}$$

   where φ_{init} is the formula that specifies the initial state of the execution, φ_{final} specifies the final state, and "$\Rightarrow$" states the rewriting/reachability relation between states (see Section 5.1).
3. There exists a matching logic *proof system* that defines the provability relation $\vdash$ between theories and formulas. For example, the correctness of the above execution from φ_{init} to φ_{final} is witnessed by the formal proof:

$$\Gamma^L \vdash \varphi_{\mathit{init}} \Rightarrow \varphi_{\mathit{final}} \tag{2}$$

Therefore, matching logic is the logical foundation of $\mathbb{K}$. The *correctness* of $\mathbb{K}$ conducting one language task is reduced to the *existence of a formal proof* in matching logic. Such formal proofs are encoded as proof objects, discussed below.

Proof Parameters. A proof parameter is the necessary information that $\mathbb{K}$ should provide to help generate proof objects. For program execution, such as Equation (2), the proof parameter includes the following information:

– the complete execution trace $\varphi_0, \varphi_1, \ldots, \varphi_n$, where $\varphi_0 \equiv \varphi_{init}$ and $\varphi_n \equiv \varphi_{\mathit{final}}$; we call $\varphi_0, \ldots, \varphi_n$ the intermediate *snapshots* of the execution;
– for each step from φ_i to φ_{i+1}, the *rewriting information* that consists of the rewrite/semantic rule $\varphi_{lhs} \Rightarrow \varphi_{rhs}$ that is applied, and the corresponding substitution θ such that $\varphi_{lhs}\theta \equiv \varphi_i$.

In other words, a proof parameter of a program execution trace contains the complete information about how such an execution is carried out by $\mathbb{K}$. The proof parameter, once generated by $\mathbb{K}$, is passed to the proof object generator to generate the corresponding proof object, discussed below.

Proof Object Generation. In our approach, a proof object is an encoding of matching logic formal proofs, such as Equation (2). Proof objects are generated by a proof object generator from the proof parameters provided by $\mathbb{K}$. At a high level, a proof object for program execution, such as Equation (2), consists of:

1. the formalization of matching logic and its provability relation $\vdash$;
2. the formalization of the formal semantics Γ^L as a logical theory, which includes axioms that specify the rewrite/semantic rules $\varphi_{lhs} \Rightarrow \varphi_{rhs}$;
3. the formal proofs of all one-step executions, i.e., $\Gamma^L \vdash \varphi_i \Rightarrow \varphi_{i+1}$ for all i;
4. the formal proof of the final proof goal $\Gamma^L \vdash \varphi_{init} \Rightarrow \varphi_{\mathit{final}}$.

Our proof objects have a *linear structure*, which implies a nice separation of concerns. Indeed, Item 1 is only about matching logic and is *not specific* to any programming languages/language tasks, so we only need to develop and proof-check it *once and for all.* Item 2 is specific to the language semantics Γ^L but is independent of the actual program executions, so it can be reused in the proof objects of various language executions for the same programming language L.

A Trustworthy Proof Checker. A proof checker is a small program that checks whether the formal proofs encoded in a proof object are correct. The proof checker is the main trust base of our work. In this paper, we use Metamath [20]—a third-party proof checking tool that is simple, fast, and trustworthy—to formalize matching logic and encode its formal proofs.

Summary. Our approach to establishing the correctness of $\mathbb{K}$ is based on its logical foundation—matching logic. We formalize language semantics as logical theories, and program executions as formulas and proof goals, whose proof objects are automatically generated and proof-checked. Our proof objects have a linear structure that allows easy reuse of their components. The key characteristics of our logical-based approach are the following:
– It is *faithful* to the real $\mathbb{K}$ implementation because proof objects are generated from proof parameters, which include all execution snapshots and the actual rewriting information, provided by $\mathbb{K}$.
– It is *practical* because proof objects are generated for each program executions on a case-by-case bases, avoiding the verification of the entire $\mathbb{K}$.

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

```
20 module IMP imports IMP-SYNTAX
21   imports DOMAINS
22   syntax KResult ::= Int
23   configuration
24    <T> <k> $PGM:Pgm </k>
25        <state> .Map </state> </T>
26   rule <k> X:Id => I ...</k>
27        <state>... X |-> I ...</state>
28   rule I1 + I2 => I1 +Int I2
29   rule I1 - I2 => I1 -Int I2
30   rule <k> X = I:Int => I ...</k>
31        <state>... X |-> (_ => I) ...</state>
32   rule {} S:Stmt => S
33   rule if(I) S _ => S requires I =/=Int 0
34   rule if(0) _ S => S
35   rule while(B) S => if(B) {S while(B) S} {}
36   rule <k> int (X, Xs => Xs) ; S </k>
37        <state>... (. => X |-> 0) </state>
38   rule int .Ids ; S => S
39 endmodule
```

Fig. 2: The complete $\mathbb{K}$ formal definition of an imperative language IMP.

– It is *trustworthy* because the autogenerated proof objects are checked using the trustworthy third-party Metamath proof checker.

3 $\mathbb{K}$ Framework and Generation of Proof Parameters

3.1 $\mathbb{K}$ Overview

$\mathbb{K}$ is an effort in realizing the ideal language framework vision in Figure 1. An easy way to understand $\mathbb{K}$ is to look at it as a meta-language that can define other programming languages. In Figure 2, we show an example $\mathbb{K}$ language definition of an imperative language IMP. In the 39-line definition, we *completely* define the formal syntax and the (executable) formal semantics of IMP, using a front end language that is easy to understand. From this language definition, $\mathbb{K}$ can generate all language tools for IMP, including its parser, interpreter, verifier, etc.

We use IMP as an example to illustrate the main $\mathbb{K}$ features. There are two *modules*: `IMP-SYNTAX` defines the syntax and `IMP` defines the semantics using rewrite rules. Syntax is defined as BNF grammars. The keyword `syntax` leads production rules that can have attributes that specify the additional syntactic and/or semantic information. For example, the syntax of `if`-statements is defined in lines 11-12 and has the attribute `[strict(1)]`, meaning that the evaluation order is strict in the first argument, i.e., the condition of an `if`-statement.

In the module `IMP`, we define the *configurations* of IMP and its formal semantics. A configuration (lines 23-25) is a constructor term that has all semantic information needed to execute programs. IMP configurations are simple, consisting of the IMP code and a program state that maps variables to values. We organize configurations using *(semantic) cells*: `</k>` is the cell of IMP code and `</state>` is the cell of program states. In the initial configuration (lines 24-25), `</state>` is empty and `</k>` contains the IMP program that we pass to $\mathbb{K}$ for execution (represented by the special $\mathbb{K}$ variable `$PGM`).

We define formal semantics using *rewrite rules*. In lines 26-27, we define the semantics of variable lookup, where we match on a variable `X` in the `</k>` cell and look up its value `I` in the `</state>` cell, by matching on the binding `X` $\mapsto$ `I`. Then, we rewrite `X` to `I`, denoted by `X` $\Rightarrow$ `I` in the `</k>` cell in line 26. Rewrite rules in $\mathbb{K}$ are similar to those in the rewrite engines such as Maude [7].

A Running Example. IMP is too complex as a running example so we introduce a simpler one: `TWO-COUNTERS`. Although simple, `TWO-COUNTERS` still uses the core features of defining formal syntax as grammars and formal semantics as rewrite rules.

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

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

`TWO-COUNTERS` is a tiny language that defines a state machine with two counters. Its computation configuration is simply a pair $\langle m, n\rangle$ of two integers m and n, and its semantics is defined by the following (conditional) rewrite rule:

$$\langle m, n\rangle \Rightarrow \langle m-1, n+m\rangle \qquad \text{if } m > 0 \tag{3}$$

Therefore, `TWO-COUNTERS` adds n by m and reduces m by 1. Starting from the initial state $\langle m, 0\rangle$, `TWO-COUNTERS` carries out m execution steps and terminates at the final state $\langle 0, m(m+1)/2\rangle$, where $m(m+1)/2 = m + (m-1) + \cdots + 1$.

3.2 Program Execution and Proof Parameters

In the following, we show a concrete program execution trace of `TWO-COUNTERS` starting from the initial state $\langle 100, 0\rangle$:

$$\langle 100, 0\rangle, \langle 99, 100\rangle, \langle 98, 199\rangle, \ldots, \langle 1, 5049\rangle, \langle 0, 5050\rangle \tag{4}$$

To make $\mathbb{K}$ generate the above execution trace, we need to follow these steps:

1. Prepare the initial state $\langle 100, 0\rangle$ in a source file, say `100.two-counters`.
2. Compile the formal semantics `TWO-COUNTERS` into a matching logic theory, explained in Section 5.
3. Use the $\mathbb{K}$ execution tool `krun` and pass the source file to it:

```
$ krun 100.two-counters --depth K
```

The option `--depth K` tells $\mathbb{K}$ to execute for `K` steps and output the (intermediate) snapshot. By letting `K` be $1, 2, \ldots$, we collect all snapshots in Equation (4).

The *proof parameter* of Equation (4) includes the additional rewriting information for each execution step. That is, we need to know the rewrite rule that is applied and the corresponding substitution. In `TWO-COUNTERS`, there is only one rewrite rule, and the substitution can be easily obtained by pattern matching, where we simply match the snapshot with the left-hand side of the rewrite rule.

Note that we regard $\mathbb{K}$ as a "black box". We are not interested in its complex internal algorithms. Instead, we hide such complexity by letting $\mathbb{K}$ generate proof parameters that include enough information for proof object generation. This way, we create a separation of concerns between $\mathbb{K}$ and proof object generation. $\mathbb{K}$ can aim at optimizing the performance of the autogenerated language tools, *without* making proof object generation more complex.

4 Matching Logic and Its Formalization

We review the syntax and proof system of matching logic—the logical foundation of $\mathbb{K}$. Then, we discuss its formalization, which is our main technical contribution and is a critical component of the proof objects we generate for $\mathbb{K}$ (see Section 2).

4.1 Matching Logic Overview

Matching logic was proposed in [24] as a means to specify and reason about programs compactly and modularly. The key concept is its formulas, called *patterns*, which are used to specify program syntax and semantics in a uniform way. Matching logic is known for its simplicity and rich expressiveness. In [23,5,6,4], the authors developed matching logic theories that capture FOL, FOL-lfp, separation logic, modal logic, temporal logics, Hoare logic, λ-calculus, type systems, etc. In Section 5, we discuss the matching logic theories that capture $\mathbb{K}$.

The *syntax* of matching logic is parametric in two sets of variables EV and SV. We call EV the set of *element variables*, denoted $x, y, \ldots$, and SV the set of *set variables*, denoted $X, Y, \ldots$.

Definition 1. *A* (matching logic) signature Σ *is a set of* (constant) symbols*. The set of* Σ-patterns*, denoted* $\textsc{Pattern}(\Sigma)$*, is inductively defined as follows:*

$$\varphi ::= x \mid X \mid \sigma \mid \varphi_1\ \varphi_2 \mid \bot \mid \varphi_1 \to \varphi_2 \mid \exists x.\, \varphi \mid \mu X.\, \varphi$$

where in $\mu X.\, \varphi$ *we require that* φ *has no negative occurrences of* X*.*

Thus, 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. We have propositional connectives $\bot$ and $\varphi_1 \to \varphi_2$, existential quantification $\exists x.\, \varphi$, and the least fixpoints $\mu X.\, \varphi$, from which the following *notations* are defined:

$$\neg\varphi \equiv \varphi \to \bot \qquad \top \equiv \neg\bot \qquad \varphi_1 \wedge \varphi_2 \equiv \neg(\neg\varphi_1 \vee \neg\varphi_2)$$
$$\varphi_1 \vee \varphi_2 \equiv \neg\varphi_1 \to \varphi_2 \qquad \forall x.\, \varphi \equiv \neg\exists x.\, \neg\varphi \qquad \nu X.\, \varphi \equiv \neg\mu X.\, \neg\varphi[\neg X/X]$$

We use $\textsc{fv}(\varphi)$ to denote the free variables of φ, and $\varphi[\psi/x]$ and $\varphi[\psi/X]$ to denote capture-free substitution. Their (usual) definitions are listed in Figure 4.

Matching logic has a *pattern matching semantics*, where a pattern φ is interpreted as the set of elements that match it. For example, $\varphi_1 \wedge \varphi_2$ is then the pattern that is matched by those matching both φ_1 and φ_2. Matching logic semantics is not needed for proof object generation, so we exile it to Appendix B.

$x[\psi/x] \equiv \psi$	$y[\psi/x] \equiv y$ if $y \not\equiv x$
$\sigma[\psi/x] \equiv \sigma$	$(\varphi_1 \to \varphi_2)[\psi/x] \equiv \varphi[\psi_1/x] \to \varphi_2[\psi/x]$
$\bot[\psi/x] \equiv \bot$	$(\varphi_1\, \varphi_2)[\psi/x] \equiv (\varphi_1[\psi/x])\ (\varphi_2[\psi/x])$
$(\exists x.\, \varphi)[\psi/x] \equiv \exists x.\, \varphi$	$(\exists x.\, \varphi)[\psi/y] \equiv \exists z.\, \varphi[z/x][\psi/y]$ for fresh z
	$(\mu X.\, \varphi)[\psi/x] \equiv \mu Z.\, \varphi[Z/X][\psi/x]$ for fresh Z

Fig. 4: Capture-free substitution are defined in the usual way and formalized later in Section 4.2 as a part of our proof objects.

We show the *matching logic proof system* Figure 5, which defines the provability relation, written $\Gamma \vdash \varphi$, which means that φ can be proved using the proof system, with patterns in Γ added as additional axioms. We call Γ a *matching logic theory*. The proof system is a main component of proof objects so we discuss it in detail below. To understand it, we first need to define *application contexts*.

Definition 2. *A* context *is a pattern C with a hole variable $\square$. We write $C[\varphi] \equiv C[\varphi/\square]$ as the result of context plugging. We call C an* application context*, if*

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 $\square \notin \mathrm{FV}(\varphi)$.*

That is, the path from the root to $\square$ in C has only applications.

The proof rules are sound (Theorem 1) and can be divided into 4 categories: FOL reasoning, frame reasoning, fixpoint reasoning, and some technical rules. The FOL reasoning rules provide (complete) FOL reasoning (see, e.g., [26]). The frame reasoning rules state that application contexts are commutative with disjunctive connectives such as $\vee$ and $\exists$. The fixpoint reasoning rules support the standard fixpoint reasoning as in modal μ-calculus [16]. The technical proof rules are needed for some completeness results (see [5] for details).

4.2 Formalizing Matching Logic

We discuss the formalization of matching logic, which is our first main contribution and forms an important component in our proof objects (see Section 2).

Metamath [20] is a tiny language to state abstract mathematics and their proofs in a machine-checkable style. In our work, we use Metamath to formalize matching logic and to encode our proof objects. We choose Metamath for its simplicity and fast proof checking: Metamath proof checkers are often hundreds lines of code and can proof-check thousands of theorems in a second.

Our formalization follows closely Section 4.1. We formalize the syntax of patterns and the proof system. We also need to formalize some metalevel operations such as free variables and capture-free substitution. An *innovative* contribution is a generic way to handling *notations* (such as $\neg$ and $\wedge$) in matching logic. The resulting formalization has only 245 lines of code, which we show in Appendix A. This formalization of matching logic is the main trust base of our proof objects.

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

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

Metamath Overview. We use an extract of our formalization of matching logic (Figure 6) to explain the basic concepts in Metamath. At a high level, a Metamath source file consists of a list of *statements*. The main ones are:

1. *constant statements* (`$c`) that declare Metamath constants;
2. *variable statements* (`$v`) that declare Metamath variables, and *floating statements* (`$f`) that declare their intended ranges;
3. *axiomatic statements* (`$a`) that declare Metamath axioms, which can be associated with some *essential statements* (`$e`) that declare the premises;
4. *provable statements* (`$p`) that states a Metamath theorem and its proof.

Figure 6 defines the fragment of matching logic with only implications. We declare five constants in a row in line 1, where `\imp` , `(` , and `)` build the syntax, `#Pattern` is the type of patterns, and `|-` is the provability relation. We declare three metavariables of patterns in lines 3-6, and the syntax of implication $\varphi_1 \to \varphi_2$ as `( \imp ph1 ph2 )` in line 7. Then, we define matching logic proof rules as Metamath axioms. For example, lines 18-22 define the rule (Modus Ponens).

```
1  $c \imp ( ) #Pattern |- $.
2
3  $v ph1 ph2 ph3 $.
4  ph1-is-pattern $f #Pattern ph1 $.
5  ph2-is-pattern $f #Pattern ph2 $.
6  ph3-is-pattern $f #Pattern ph3 $.
7  imp-is-pattern
8    $a #Pattern ( \imp ph1 ph2 ) $.
9
10 axiom-1
11   $a |- ( \imp ph1 ( \imp ph2 ph1 ) ) $.
12
13 axiom-2
14   $a |- ( \imp ( \imp ph1 ( \imp ph2 ph3 ) )
15         ( \imp ( \imp ph1 ph2 )
16                ( \imp ph1 ph3 ) ) ) $.
17
18 ${
19   rule-mp.0 $e |- ( \imp ph1 ph2 ) $.
20   rule-mp.1 $e |- ph1 $.
21   rule-mp   $a |- ph2 $.
22 $}
23 imp-refl $p |- ( \imp ph1 ph1 )
24 $=
25   ph1-is-pattern ph1-is-pattern
26   ph1-is-pattern imp-is-pattern
27   imp-is-pattern ph1-is-pattern
28   ph1-is-pattern imp-is-pattern
29   ph1-is-pattern ph1-is-pattern
30   ph1-is-pattern imp-is-pattern
31   ph1-is-pattern imp-is-pattern
32   imp-is-pattern ph1-is-pattern
33   ph1-is-pattern ph1-is-pattern
34   imp-is-pattern imp-is-pattern
35   ph1-is-pattern ph1-is-pattern
36   imp-is-pattern imp-is-pattern
37   ph1-is-pattern ph1-is-pattern
38   ph1-is-pattern imp-is-pattern
39   ph1-is-pattern axiom-2
40   ph1-is-pattern ph1-is-pattern
41   ph1-is-pattern imp-is-pattern
42   axiom-1 rule-mp ph1-is-pattern
43   ph1-is-pattern axiom-1 rule-mp
44 $.
```

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

In line 23, we show an example (meta-)theorem and its formal proof in Metamath. The theorem states that $\vdash \varphi_1 \to \varphi_1$ holds, and its proof (lines 25-43) is a sequence of labels referring to the previous axiomatic/provable statements.

Metamath proofs are very easy to proof-check, which is why we use it in our work. The proof checker reads the labels in order and push them to a *proof stack* S, which is initially empty. When a label l is read, the checker pops its premise statements from S and pushes l itself. When all labels are consumed, the checker checks whether S has exactly one statement, which should be the original proof goal. If so, the proof is checked. Otherwise, it fails.

As an example, we look at the first 5 labels of the proof in Figure 6, line 25:

```
                 // Initially, the proof stack S is empty
ph1-is-pattern   // S = [ #Pattern ph1 ]
ph1-is-pattern   // S = [ #Pattern ph1 ; #Pattern ph1 ]
ph1-is-pattern   // S = [ #Pattern ph1 ; #Pattern ph1 ; #Pattern ph1 ]
imp-is-pattern   // S = [ #Pattern ph1 ; #Pattern ( \imp ph1 ph1 ) ]
imp-is-pattern   // S = [ #Pattern ( \imp ph1 ( \imp ph1 ph1 ) ) ]
```

where we show the stack status in comments. The first label `ph1-is-pattern` refers to a `$f`-statement without premises, so nothing is popped off, and the corresponding statement `#Pattern ph1` is pushed to the stack. The same happens, for the second and third labels. The fourth label `imp-is-pattern` refers to a `$a`-statement with two metavariables of patterns, and thus has 2 premises. Therefore, the top two statements in S are popped off, and the corresponding conclusion `#Pattern ( \imp ph1 ph1 )` is pushed to S. The last label does the same, popping off two premises and pushing `#Pattern ( \imp ph1 ( \imp ph1 ph1 ) )` to S. Thus, these five proof steps prove the *wellformedness* of $\varphi_1 \to (\varphi_1 \to \varphi_1)$.

Formalizing Matching Logic Syntax. Now, we go through the formalization of matching logic and emphasize some highlights. See Appendix A for full details.

The syntax of patterns is formalized below, following Definition 1:

```
$c \bot \imp \app \exists \mu ( ) $.
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 ) $.  $}
```

Note that we omit the declarations of metavariables (such as `xX`, `sg0`, ...) because their meaning can be easily inferred. The only nontrivial case above is `mu-is-pattern`, where we require that `ph0` is positive in `X`, discussed below.

Metalevel Assertions. To formalize matching logic, we need the following metalevel operations and/or assertions:

1. positive (and negative) occurrences of variables;
2. free variables;
3. capture-free substitution;
4. application contexts;
5. notations.

Item 1 is needed to define the syntax of $\mu X.\,\varphi$, while Items 2-5 are needed to define the proof system (Figure 5). Here, we show how to define capture-free substitution as an example. Notations are discussed in the next section.

To formalize capture-free substitution, we first define a Metamath constant

```
$c #Substitution $.
```

that serves as an assertion symbol: `#Substitution ph ph' ph" xX` holds iff `ph` $\equiv$ `ph'` [`ph"` / `xX`]. Then, we can define substitution following Figure 4. The only nontrivial case is when `ph'` is $\exists x.\,\varphi$ or $\mu X.\,\varphi$, in which case α-renaming is required to avoid variable capture. We show the case when `ph'` is $\exists x.\,\varphi$ below:

```
substitution-exists-shadowed
   $a #Substitution ( \exists x ph1 ) ( \exists x ph1 ) ph0 x $.
${ $d xX x  $.
   $d y ph0 $.
   substitution-exists.0 $e #Substitution ph2 ph1 y x $.
   substitution-exists.1 $e #Substitution ph3 ph2 ph0 xX $.
   substitution-exists
     $a #Substitution ( \exists y ph3 ) ( \exists x ph1 ) ph0 xX $. $}
```

There are two cases, as expected from Figure 4. `substitution-exists-shadowed` is when the substitution is shadowed. `substitution-exists` is the general case, where we first rename `x` to a fresh variable `y` and then continue the substitution. The `$d`-statements state that the substitution is not shadowed and `y` is fresh.

Supporting Notations. Notations (e.g., $\neg$ and $\wedge$) play an important role in matching logic. Many proof rules such as (Propagation$_\vee$) and (Singleton) use notations (see Figure 5). However, Metamath has no built-in support for notations. To define a notation, say $\neg\varphi \equiv \varphi \to \bot$, we need to (1) declare a constant `\not` and add it to the pattern syntax; (2) define the equivalence relation $\neg\varphi \equiv \varphi \to \bot$; and (3) add a new case for `\not` to *every metalevel assertions*. While (1) and (2) are reasonable, we want to avoid (3) because there are many metalevel assertions and thus it creates duplication.

Therefore, we implement an innovative and generic method that allows us to define *any notations* in a compact way. Our method is to declare a new constant `#Notation` and use it to capture the *congruence relation of sugaring/desugaring*. Using `#Notation`, it takes only three lines to define the notation $\neg\varphi \equiv \varphi \to \bot$:

```
$c \not $.
not-is-pattern $a #Pattern  ( \not ph0 ) $.
not-is-sugar   $a #Notation ( \not ph0 ) ( \imp ph0 \bot ) $.
```

To make the above work, we need to state that `#Notation` is a congruence relation with respect to the syntax of patterns and all the other metalevel assertions. Firstly, we state that it is reflexive, symmetric, and transitive:

```
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 $. $}
```

And the following is an example where we state that `#Notation` is a congruence with respect to provability:

```
${ notation-provability.0 $e #Notation ph0 ph1 $.
   notation-provability.1 $e |- ph0 $.
   notation-provability   $a |- ph1 $. $}
```

This way, we only need a *fixed* number of statements that state that `#Notation` is a congruence, making it more compact and less duplicated to define notations.

Formalizing Proof System. With metalevel assertions and notations, it is now straightforward to formalize matching logic proof rules. We have seen the formalization of (Modus Ponens) in Figure 6. In the following, we formalize the fixpoint proof rule (Kanaster-Tarski), whose premises use capture-free substitution:

```
${ rule-kt.0 $e #Substitution ph0 ph1 ph2 X $.
   rule-kt.1 $e |- ( \imp ph0 ph2 ) $.
   rule-kt   $a |- ( \imp ( \mu X ph1 ) ph2 ) $. $}
```

5 Compiling $\mathbb{K}$ into Matching Logic

To execute programs using $\mathbb{K}$, we need to compile the $\mathbb{K}$ language definition for language L into a matching logic theory, written Γ^L (see Section 3.2). In this section, we discuss this compilation process and show how to formalize Γ^L.

5.1 Basic Matching Logic Theories

Firstly, we discuss the basic matching logic theories that are required by Γ^L. We discuss the theories of equality, sorts (and sorted functions), and rewriting.

Theory of Equality. By equality, we mean a (predicate) pattern $\varphi_1 = \varphi_2$ that holds (i.e., equals to $\top$) iff φ_1 equals to φ_2, and fails (i.e., equals to $\bot$) otherwise. We first need to define *definedness* $\lceil \varphi \rceil$, which is a predicate pattern that states that φ is *defined*, i.e., φ is matched by at least one element: φ is not $\bot$.

Definition 3. *Consider a symbol* $\lceil \_ \rceil \in \Sigma$*, called the* definedness symbol*. We write* $\lceil \varphi \rceil$ *for the application* $\lceil \_ \rceil\ \varphi$*. In addition, we define the following axiom:*

$$\text{(Definedness)} \quad \lceil x \rceil \tag{5}$$

(Definedness) states that any element x is *defined*. Using the definedness symbol, we can define many important mathematical instruments, including equality, as the following notations:

$$\begin{array}{llll} \lfloor \varphi \rfloor \equiv \neg \lceil \neg \varphi \rceil & // \text{ Totality} & \varphi_1 = \varphi_2 \equiv \lfloor \varphi_1 \leftrightarrow \varphi_2 \rfloor & // \text{ Equality} \\ \varphi_1 \subseteq \varphi_2 \equiv \lfloor \varphi_1 \to \varphi_2 \rfloor & // \text{ Inclusion} & x \in \varphi \equiv \lceil x \wedge \varphi \rceil & // \text{ Membership} \end{array}$$

Proposition 1 shows that the above indeed capture the intended semantics.

Theory of Sorts. Matching logic is not sorted, but $\mathbb{K}$ is. To compile $\mathbb{K}$ into matching logic, we need a systematic way to dealing with sorts. We follow the "sort-as-predicate" paradigm to handle sorts and sorted functions in matching logic, following [6,4]. The main idea is to define a symbol $\llbracket \_ \rrbracket \in \Sigma$, called the *inhabitant symbol*, and use the *inhabitant pattern* $\llbracket s \rrbracket$ (abbreviated for the application $\llbracket \_ \rrbracket\, s$) to represent the *inhabitant set* of sort s. For example, to define a sort Nat, we define a corresponding symbol Nat that represents the sort name, and use $\llbracket Nat \rrbracket$ to represent the set of all natural numbers.

Sorted functions can be axiomatized as special matching logic symbols. For example, the successor function $succ$ of natural numbers is a symbol with axiom:

$$\forall x.\, x \in \llbracket Nat \rrbracket \to \exists y.\, y \in \llbracket Nat \rrbracket \wedge succ\, x = y \tag{6}$$

In other words, for any x in the inhabitant set of Nat, there exists a y in the inhabitant set of Nat such that $succ\, x$ equals to y. Thus, $succ$ is a sorted function from Nat to Nat.

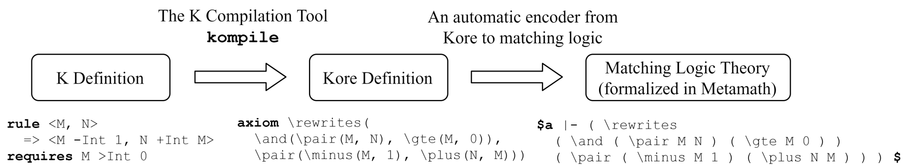

Fig. 7: Automatic translation from $\mathbb{K}$ to matching logic, via Kore

Theory of Rewriting. Recall that in $\mathbb{K}$, the formal language semantics is defined using rewrite rules, which essentially define a *transition system* over computation configurations. In matching logic, a transition system can be captured by only one symbol $\bullet \in \Sigma$, called *one-path next*, with the intuition that for any configuration γ, $\bullet\gamma$ is matched by all configurations that can go to γ in one step. In other words, γ is reached on *one-path* in the *next* configuration.

Program execution is the reflexive and transitive closure of one-path next. Formally, we define program execution (i.e., rewriting) as follows:

$$\begin{aligned} \diamond\varphi &\equiv \mu X.\, \varphi \vee \bullet X && \text{// Eventually; equals to } \varphi \vee \bullet\varphi \vee \bullet\bullet\varphi \vee \dots \\ \varphi_1 \Rightarrow \varphi_2 &\equiv \varphi_1 \to \diamond\varphi_2 && \text{// Rewriting} \end{aligned}$$

5.2 Kore: The Intermediate Between $\mathbb{K}$ and Matching Logic

The $\mathbb{K}$ compilation tool `kompile` (explained shortly) is what compiles a $\mathbb{K}$ language definition into the matching logic theory Γ^L, written in a formal language called Kore. For legacy reasons, the Kore language is not the same as the syntax of matching logic (Definition 1), but is an axiomatic extension with equality, sorts, sorted functions, and rewriting. Therefore, to formalize Γ^L in our proof objects, we need to (1) formalize the basic matching logic theories that we showed above; and (2) automatically translate Kore definitions into the corresponding matching logic theories. Due to space limit, we exile details about Kore to Appendix C. Instead, we use Figure 7 to show the 2-phase translation from $\mathbb{K}$ to matching logic, via Kore.

Phase 1: From $\mathbb{K}$ to Kore. To compile a $\mathbb{K}$ definition such as `two-counters.k` in Figure 3, we pass it to the $\mathbb{K}$ compilation tool `kompile` as follows:

```
$ kompile two-counters.k
```

The result is a compiled Kore definition `two-counters.kore`. We show the auto-generated Kore axiom in Figure 7 that corresponds to the rewrite rule in Equation (3). As we can see, Kore is a much lower-level language than $\mathbb{K}$, where the programming language concrete syntax and $\mathbb{K}$'s front end syntax are parsed and replaced by the abstract syntax trees, represented by the constructor terms.

Phase 2: From Kore to Matching Logic. We develop an automatic encoder that translates Kore syntax into matching logic patterns. Since Kore is essentially the theory of equality, sorts, and rewriting, we can define the syntactic constructs of the Kore language as *notations*, using the basic theories in Section 5.1.

6 Generating Proof Objects for Program Execution

In this section, we discuss how to generate proof objects for program execution, based on the formalization of matching logic and $\mathbb{K}$/Kore in Sections 4 and 5. The key step is to generate proof objects for *one-step executions*, which are then put together to build the proof objects for multi-step executions using the transitivity of the rewriting relation. Thus, we focus on the process of generating proof objects for one-step executions from the proof parameters provided by $\mathbb{K}$.

6.1 Problem Formulation

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

$$S = \{t_k \wedge p_k \Rightarrow s_k \mid k = 1, 2, \ldots, K\}$$

where t_k and s_k are the left- and right-hand sides of the rewrite rule, respectively, and p_k is the rewriting condition. Consider the following execution trace:

$$\varphi_0, \varphi_1, \ldots, \varphi_n \tag{7}$$

where $\varphi_0, \ldots, \varphi_n$ are snapshots. We let $\mathbb{K}$ generate the following proof parameter:

$$\Theta \equiv (k_0, \theta_0), \ldots, (k_{n-1}, \theta_{n-1}) \tag{8}$$

where for each $0 \leq i < n$, k_i denotes the rewrite rule that is applied on φ_i ($1 \leq k_i \leq K$) and θ_i denotes the corresponding substitution such that $t_{k_i}\theta_i = \varphi_i$.

As an example, the rewrite rule of `TWO-COUNTERS`, restated below:

$$\langle m, n \rangle \Rightarrow \langle m-1, n+m \rangle \quad \text{if } m > 0 \qquad \text{// Same as Equation (3)}$$

has the left-hand side $t_k \equiv \langle m, n \rangle$, the right-hand side $s_k \equiv \langle m-1, n+m \rangle$, and the condition $p_k \equiv m \geq 0$. Note that the right-hand side pattern s_k contains the arithmetic operations "+" and "−" that can be further evaluated to a value, if concrete instances of the variables m and n are given. Generally speaking, the right-hand side of a rewrite rule may include (built-in or user-defined) functions that are not constructors and thus can be further evaluated. We call such evaluation process a *simplification*.

6.2 Applying Rewrite Rules and Applying Simplifications

In the following, we list all proof objects for one-step executions.

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

As we can see, there are two types of proof objects: one that proves the results of *applying rewrite rules* and one that *applies simplification*.

Applying Rewrite Rules. The main steps in proving $\Gamma^L \vdash \varphi_i \Rightarrow s_{k_i}\theta_i$ are (1) to *instantiate* the rewrite rule $t_{k_i} \wedge p_{k_i} \Rightarrow s_{k_i}$ using the substitution

$$\theta_i = [c_1/x_1, \ldots, c_m/x_m]$$

given in the proof parameter, and (2) to show that the (instantiated) rewriting condition $p_{k_i}\theta_i$ holds. Here, $x_1, \ldots, x_m$ are the variables that occur in the rewrite rule and $c_1, \ldots, c_m$ are terms by which we instantiate the variables. For (1), we need to first prove the following lemma, called (Functional Substitution) in [5], which states that $\forall$-quantification can be instantiated by functional patterns:

$$\frac{\forall \vec{x}.\, t_{k_1} \wedge p_{k_i} \Rightarrow s_{k_i} \quad \exists y_1.\, \varphi_1 = y_1 \quad \cdots \quad \exists y_m.\, \varphi_m = y_m}{t_{k_i}\theta_i \wedge p_{k_i}\theta_i \Rightarrow s_{k_i}\theta_i} \; y_1, \ldots, y_m \text{ fresh}$$

Intuitively, the premise $\exists y_1.\, \varphi_1 = y_1$ states that φ_1 is a functional pattern because it equals to some element y_1.

If Θ in Equation (8) is the correct proof parameter, θ_i is the correct substitution and thus $t_{k_i}\theta_i \equiv \varphi_i$. Therefore, to prove the original proof goal for one-step execution, i.e. $\Gamma^L \vdash \varphi_i \Rightarrow s_{k_i}\theta_i$, we only need to prove that $\Gamma^L \vdash p_{k_i}\theta_i$, i.e., the rewriting condition p_{k_i} holds under θ_i. This is done by *simplifying* $p_{k_i}\theta_i$ to $\top$, discussed together with the simplification process in the following.

Applying Simplifications. $\mathbb{K}$ carries out simplification exhaustively before trying to apply a rewrite rule, and simplifications are done by applying (oriented) equations. Generally speaking, let s be a functional pattern and $p \to t = t'$ be a (conditional) equation, we say that s can be *simplified* w.r.t. $p \to t = t'$, if there is a sub-pattern s_0 of s (written $s \equiv C[s_0]$ where C is a context) and a substitution θ such that $s_0 = t\theta$ and $p\theta$ holds. The resulting *simplified pattern* is denoted $C[t'\theta]$. Therefore, a proof object of the above simplification consists of two proofs: $\Gamma^L \vdash s = C[t'\theta]$ and $\Gamma^L \vdash p\theta$. The latter can be handled recursively, by simplifying $p\theta$ to $\top$, so we only need to consider the former.

The main steps of proving $\Gamma^L \vdash s = C[t'\theta]$ are the following:

1. to find C, s_0, θ, and $t = t'$ in Γ^L such that $s \equiv C[s_0]$ and $s_0 = t\theta$; in other words, s can be simplified w.r.t. $t = t'$ at the sub-pattern s_0;
2. to prove $\Gamma^L \vdash s_0 = t'\theta$ by instantiating $t = t'$ using the substitution θ, using the same (Functional Substitution) lemma as above;
3. to prove $\Gamma^L \vdash C[s_0] = C[t']$ using the transitivity of equality, which is proved as part of the theory of equality (see Proposition 2).

Finally, we repeat the above one-step simplifications until no sub-patterns can be simplified further. The resulting proof objects are then put together by the transitivity of equality.

7 Discussion on Implementation

As discussed in Section 2, a complete proof object for program execution (i.e., $\Gamma^L \vdash \varphi_{init} \Rightarrow \varphi_{final}$) consists of (1) the formalization of matching logic and its basic theories; (2) the formalization of Γ^L; and (3) the proofs of one-step and multi-step program executions. In our implementation, (1) is developed manually because it is fixed for all programming languages and program executions. (2) and (3) are automatically generated by the algorithms in Section 6.

During the (manual) development of (1), we needed to prove many basic matching logic (meta-)theorems as lemmas, such as (Functional Substitution) in Section 6.2. To ease the manual work, we developed an *interactive theorem prover* (ITP) for matching logic, which allows us to carry out higher-level interactive proofs that are later automatically translated into the lower-level Metamath proofs. We show the highlights of our ITP for matching logic in Section 7.1.

In Section 7.2, we discuss the main limitations of our current preliminary implementation. These limitations are planned to be addressed in future work.

7.1 An Interactive Theorem Prover for Matching Logic

Metamath proofs are low-level and not human readable (see, e.g., the proof of $\vdash \varphi \to \varphi$ in Figure 6). Metamath has its own interactive theorem prover (ITP), but it is for general purposes and does not have specific for matching logic. Therefore, we developed a new ITP for matching logic that has the following characteristic features:

- Our ITP understands the syntax of matching logic patterns and has proof tactics to *desugar* notations in the proof goals;
- Our ITP has an automatic proof tactic for propositional tautologies, based on the *resolution* method;
- Our ITP allows *dynamic proofs*, meaning that new lemmas can be dynamically added during an interactive proof; this makes our ITP easier to use.

When an interactive proof is finished, our ITP will translate the higher-level proof tactics into real Metamath formal proofs, and thus ease the manual development. Due to space limit, we exile the details about our ITP to Appendix D.

7.2 Limitations and Threats to Validity

We discuss the trust base of the autogenerated proof objects by pointing out the main threats to validity, caused by the limitations of our preliminary implementation. It should be noted that these limitations are about the implementation, and *not* our approach. We shall address these limitations in future work.

Limitation 1: Need to trust Kore. Our current implementation is based on the existing $\mathbb{K}$ compilation tool `kompile` that compiles $\mathbb{K}$ into Kore definitions. Recall that Kore is a (legacy) formal language with built-in support for equality, sorts, and rewriting, and thus is different (and more complex) than the syntax of matching logic. By using Kore as the intermediate between $\mathbb{K}$ and matching logic (Figure 7), we need to trust Kore and the $\mathbb{K}$ complication tool `kompile`.

In the future, we will eliminate Kore entirely from the picture and formalize $\mathbb{K}$ *directly*. To do that, we need to formalize the "front end matters" of $\mathbb{K}$, such as concrete programming language syntax and $\mathbb{K}$ attributes, currently handled by `kompile`. That is, we need to formalize and generate proof objects for `kompile`.

Limitation 2: Need to trust domain reasoning. $\mathbb{K}$ has built-in support for domain reasoning such as integer arithmetic. Our current proof objects do not include the formal proofs of such domain reasoning, but instead regard them as assumed lemmas. In the future, we will incorporate the existing research on generating proof objects for SMT solvers [2] into our implementation, in order to generate proof objects also for domain reasoning; see also Section 9.

Limitation 3: Do not support more complex $\mathbb{K}$ features. Our current implementation only supports the core $\mathbb{K}$ features of defining programming language syntax and of defining formal semantics as rewrite rules. Some more complex features are not supported; the main ones are (1) the `[strict]` attributes that specify evaluation orders; and (2) the use of built-in collection datatypes, such as lists, sets, and maps.

To support (1), we should handle the so-called *heating/cooling rules* that are autogenerated rewrite rules that implement the specified evaluation orders. Our current implementation does not support these heating/cooling rules because they are conditional rules, and their conditions are those that state that an element is *not* a computation result. To prove such conditions, we need additional constructors axioms sorts that represent results of computation. To support (2), we should extend our algorithms in Section 6 with *unification* modulo these collection datatypes.

8 Evaluation

In this section, we evaluate the performance of our implementation and discuss the experiment results, summarized in Table 1. We use two sets of benchmarks. The first is our running example `TWO-COUNTERS` with different inputs (10, 20, 50, and 100). The second is REC [11], which is a popular performance benchmark for rewriting engines. We evaluate both the performance of proof object *generation* and that of proof *checking*. Our implementation can be found in [1].

The main takeaways of our experiments are:

1. Proof checking is efficient and takes a few seconds; in particular, the *task-specific* checking time is often less than one second ("task" column in Table 1).
2. Proof object generation is slower and takes several minutes.
3. Proof objects are huge, often of millions LOC (wrapped at 80 characters).

Proof Object Generation. We measure the proof object generation time as the time to generate complete proof objects following the algorithms in Section 6, from the compiled language semantics (i.e., Kore definitions) and proof

Table 1: Performance of proof object generation and proof checking.

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

parameters. As shown in Table 1, proof generation takes around 17–406 seconds on the benchmarks, and the average is 107 seconds.

Proof object generation can be divided into two parts: that of the language semantics Γ^L and that of the (one-step and multi-step) program executions. Both parts are shown in Table 1 under columns “sem” and “rewrite”, respectively. For the same language, the time to generate language semantics Γ^L is the same (up to experimental error). The time for executions is linear to the number of steps.

Proof Checking. Proof checking is efficient and takes a few seconds on our benchmarks. We can divide the proof checking time into two parts: that of the logical foundation and that of the actual program execution tasks. Both parts are shown in Table 1 under columns “logic” and “task”. The “logic” part includes formalization of matching logic and its basic theories, and thus is *fixed* for any programming language and program and has the same proof checking time (up to experiment error). The “task” part includes the language semantics and proof objects for the one-step and multi-step executions. Therefore, the time to check the “task” part is a *more valuable and realistic* measure, and according to our experiments, it is often less than 1 second, making it acceptable in practice.

As a pleasant surprise, the time for “task-specific”proof checking is roughly the same as the time that it takes $\mathbb{K}$ to parse and execute the programs. In other words, there is *no significant performance difference* on our benchmarks between running the programs directly in $\mathbb{K}$ and checking the proof objects.

There exists much potential to optimize the performance of proof checking and make it even faster than program execution. For example, in our approach proof checking is an *embarrassingly parallel problem*, because each meta-theorems can be proof-checked entirely independently. Therefore, we can significantly reduce the proof checking time by running the multiple checkers in parallel.

9 Related Work and Conclusion

The idea of using proof generation to address the functional correctness of complicated systems has been introduced a long time ago.

Interactive theorem provers such as Coq [18] and Isabelle [27] are often used to formalize programming language semantics and to reason about program properties. These provers often provide a high-level proof script language that allows the users to develop human-readable proofs, which are then automatically translated into lower-level proof objects that can be checked by the corresponding proof checkers. For example, the proof objects of Coq are of the form $t : t'$, where t' is a term that represents the proposition to be proved and t' represents a formal proof. The typing claim $t : t'$ can then be proof-checked by a proof checker that implements the typing rules of the calculus of inductive constructions (CIC) [8], which is the logical foundation of Coq.

There are two main differences between provers such as Coq and our technique. Firstly, Coq is not regarded as a language framework in the sense of Figure 1 because no language tools are autogenerated from the formal semantics. In our case, we need to be able to handle the correctness of individual tasks on a case-by-case basis to reduce the complexity. Secondly, Coq proof checking is based on CIC, which is arguably more complex than matching logic—the logical foundation of $\mathbb{K}$ as demonstrated in this paper. Indeed, the formalization of matching logic requires only 245 LOC which we display entirely in Appendix A.

Another application of proof generation is to ensure the correctness of SMT solvers. These are popular tools to check the satisfiability of FOL formulas, written in a formal language containing interpreted functions and predicates. SMT solvers often implement complex data structures and algorithms, putting their correctness at risk. There is recent work such as [2] studying proof generation for SMT solvers. The research has been incorporated in theorem provers such as Lean, which attempts to bridge the gap between SMT reasoning and proof assistants more directly by building a proof assistant with efficient and sophisticated built-in SMT capabilities. As discussed in Section 7, our current implementation does not generate proofs for domain reasoning. So, we plan to incorporate the above SMT proof generation work into our future implementation.

Conclusion. We propose an innovative approach based on proof generation. The key idea is to generate proof objects as *proof certificates* for each individual task that the language tools conduct, on a case-by-case basis. This way, we avoid formally verifying the entire framework, which is practically impossible, and thus can make the language framework both *practical* and *trustworthy*.

References

1. Anonymous Authors. Proof object generation for K. Github project `https://github.com/cav2021-paper-1/cav2021-paper-1`.
2. Clark Barrett, Leonardo De Moura, and Pascal Fontaine. Proofs in satisfiability modulo theories. *All about proofs, Proofs for all*, 55(1):23–44, 2015.
3. Denis Bogdanas and Grigore Roşu. K-java: A complete semantics of java. In *Proceedings of the 42^{nd} Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 445–456, 2015.
4. Xiaohong Chen, Dorel Lucanu, and Grigore Roşu. Initial algebra semantics in matching logic. Technical Report http://hdl.handle.net/2142/107781, University of Illinois at Urbana-Champaign, July 2020.
5. Xiaohong Chen and Grigore Roşu. Matching μ-logic. In *Proceedings of the 34^{th} Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'19)*, pages 1–13, Vancouver, Canada, 2019. IEEE.
6. Xiaohong Chen and Grigore Roşu. A general approach to define binders using matching logic. In *Proceedings of the 25^{th} ACM SIGPLAN International Conference on Functional Programming (ICFP'20)*, pages 1–32, New Jersey, USA, 2020. ACM.
7. Manuel Clavel, Francisco Durán, Steven Eker, Santiago Escobar, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, Rubén Rubio, and Carolyn Talcott. *Maude manual (version 3.0)*. SRI International, 2020.
8. Coq Team. Coq documents: calculus of inductive constructions. Online at `https://coq.inria.fr/refman/language/cic.html`., 2020.
9. Andrei Ştefănescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore Roşu. Semantics-based program verifiers for all languages. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16)*, pages 74–91. ACM, 2016.
10. Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S. Adve, and Grigore Roşu. A complete formal semantics of x86-64 user-level instruction set architecture. In *Proceedings of the 40^{th} ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19)*, pages 1133–1148, Phoenix, Arizona, USA, 2019. ACM.
11. Francisco Durán and Hubert Garavel. The rewrite engines competitions: a RECtrospective. In Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–100, Cham, 2019. Springer International Publishing.
12. Chucky Ellison and Grigore Rosu. An executable formal semantics of c with applications. *ACM SIGPLAN Notices*, 47(1):533–544, 2012.
13. Dwight Guth. A formal semantics of python 3.3. 2013.
14. Dwight Guth, Chris Hathhorn, Manasvi Saxena, and Grigore Roşu. RV-Match: Practical semantics-based program analysis. In *Proceedings of the 28^{th} International Conference on Computer Aided Verification (CAV'16)*, volume 9779, pages 447–453. Springer, 2016.
15. Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, Brandon Moore, Yi Zhang, Daejun Park, Andrei Ştefănescu, and Grigore Roşu. KEVM: A complete semantics of the Ethereum virtual machine. In *Proceedings of the 2018 IEEE Computer Security Foundations Symposium (CSF'18)*, pages 204–217, Oxford, UK, 2018. IEEE. `http://jellopaper.org`.

16. Dexter Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27(3):333–354, 1983.
17. Qingzhou Luo, Yi Zhang, Choonghwan Lee, Dongyun Jin, Patrick O'Neil Meredith, Traian Florin Şerbănuţă, and Grigore Roşu. RV-Monitor: Efficient parametric runtime verification with simultaneous properties. In *Proceedings of the 5th International Conference on Runtime Verification (RV'14)*, pages 285–300, 2014.
18. Coq Team. *The Coq proof assistant*. LogiCal Project, 2020.
19. Norman Megill. set.mm: ZFC set theory in metamath. `https://github.com/metamath/set.mm`, 2021.
20. Norman Megill and David A. Wheeler. *Metamath: a computer language for mathematical proofs*. Lulu. com, 2019.
21. Daejun Park, Andrei Ştefănescu, and Grigore Roşu. KJS: A complete formal semantics of JavaScript. In *Proceedings of the 36th annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*, pages 346–356, Portland, OR, 2015. ACM.
22. John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1):23–41, 1965.
23. Grigore Roşu. Matching logic. *Logical Methods in Computer Science*, 13(4):1–61, 2017.
24. Grigore Roşu and Wolfram Schulte. Matching logic—extended report. Technical Report Department of Computer Science UIUCDCS-R-2009-3026, University of Illinois at Urbana-Champaign, January 2009.
25. Grigore Rosu. K—A semantic framework for programming languages and formal analysis tools. In *Dependable Software Systems Engineering*. IOS Press, 2017.
26. Joseph R. Shoenfield. *Mathematical logic*. Addison-Wesley Pub. Co, 1967.
27. The Isabelle development team. Isabelle, 2018. `https://isabelle.in.tum.de/`.

A Complete Formalization of Matching Logic

We present here our formalization of matching logic. Theories of equality, sort, and rewriting are not included.

```
$c #Pattern #ElementVariable #SetVariable #Variable #Symbol $.
$c #Positive #Negative #Fresh #ApplicationContext #Substitution #Notation |- $.
$c \bot \imp \app \exists \mu ( ) $.

$v ph0 ph1 ph2 ph3 ph4 ph5 x y X Y xX yY sg0 $.

$( 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 $.
X-is-element-var $f #SetVariable X $.
Y-is-element-var $f #SetVariable Y $.
xX-is-var        $f #Variable xX $.
yY-is-var        $f #Variable yY $.
sg0-is-symbol    $f #Symbol sg0 $.

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 ) $. $}

$( Positive occurrence $)

positive-in-var    $a #Positive xX yY $.
positive-in-symbol $a #Positive xX sg0 $.
positive-in-bot    $a #Positive xX \bot $.
${  positive-in-imp.0 $e #Negative xX ph0 $.
    positive-in-imp.1 $e #Positive xX ph1 $.
    positive-in-imp   $a #Positive xX ( \imp ph0 ph1 ) $. $}
${  positive-in-app.0 $e #Positive xX ph0 $.
    positive-in-app.1 $e #Positive xX ph1 $.
    positive-in-app   $a #Positive xX ( \app ph0 ph1 ) $. $}
${  positive-in-exists.0 $e #Positive xX ph0 $.
    positive-in-exists   $a #Positive xX ( \exists x ph0 ) $. $}
${  positive-in-mu.0 $e #Positive xX ph0 $.
    positive-in-mu   $a #Positive xX ( \mu X ph0 ) $. $}
```

```
${  $d xX ph0 $.
    positive-disjoint $a #Positive xX ph0 $. $}

$( Negative occurrence $)

negative-in-symbol $a #Negative xX sg0 $.
negative-in-bot $a #Negative xX \bot $.
${  $d xX yY $.
    negative-in-var $a #Negative xX yY $. $}
${  negative-in-imp.0 $e #Positive xX ph0 $.
    negative-in-imp.1 $e #Negative xX ph1 $.
    negative-in-imp   $a #Negative xX ( \imp ph0 ph1 ) $. $}
${  negative-in-app.0 $e #Negative xX ph0 $.
    negative-in-app.1 $e #Negative xX ph1 $.
    negative-in-app   $a #Negative xX ( \app ph0 ph1 ) $. $}
${  negative-in-exists.0 $e #Negative xX ph0 $.
    negative-in-exists   $a #Negative xX ( \exists x ph0 ) $. $}
${  negative-in-mu.0 $e #Negative xX ph0 $.
    negative-in-mu   $a #Negative xX ( \mu X ph0 ) $. $}
${  $d xX ph0 $.
    negative-disjoint $a #Negative xX ph0 $. $}

$( Free variable $)

fresh-in-symbol          $a #Fresh xX sg0 $.
fresh-in-bot             $a #Fresh xX \bot $.
fresh-in-exists-shadowed $a #Fresh x ( \exists x ph0 ) $.
fresh-in-mu-shadowed     $a #Fresh X ( \mu X ph0 ) $.
${  $d xX ph0 $.
    fresh-disjoint $a #Fresh xX ph0 $. $}
${  fresh-in-imp.0 $e #Fresh xX ph0 $.
    fresh-in-imp.1 $e #Fresh xX ph1 $.
    fresh-in-imp   $a #Fresh xX ( \imp ph0 ph1 ) $. $}
${  fresh-in-app.0 $e #Fresh xX ph0 $.
    fresh-in-app.1 $e #Fresh xX ph1 $.
    fresh-in-app   $a #Fresh xX ( \app ph0 ph1 ) $. $}
${  $d xX x $.
    fresh-in-exists.0 $e #Fresh xX ph0 $.
    fresh-in-exists   $a #Fresh xX ( \exists x ph0 ) $. $}
${  $d xX X $.
    fresh-in-mu $a #Fresh xX ( \mu X ph0 ) $. $}
${  fresh-in-substitution.0 $e #Fresh xX ph1 $.
    fresh-in-substitution.1 $e #Substitution ph2 ph0 ph1 xX $.
    fresh-in-substitution   $a #Fresh xX ph2 $. $}

$( Substitution $)

substitution-var-same        $a #Substitution ph0 xX ph0 xX $.
substitution-symbol          $a #Substitution sg0 sg0 ph0 xX $.
substitution-bot             $a #Substitution \bot \bot ph0 xX $.
substitution-identity        $a #Substitution ph0 ph0 xX xX $.
substitution-exists-shadowed $a #Substitution ( \exists x ph1 ) ( \exists x ph1 ) ph0
      x $.
substitution-mu-shadowed     $a #Substitution ( \mu X ph1 ) ( \mu X ph1 ) ph0 X $.
```

```
${  substitution-imp.0  $e #Substitution ph1 ph3 ph0 xX $.
    substitution-imp.1  $e #Substitution ph2 ph4 ph0 xX $.
    substitution-imp    $a #Substitution ( \imp ph1 ph2 ) ( \imp ph3 ph4 ) ph0 xX $.
         $}
${  substitution-app.0  $e #Substitution ph1 ph3 ph0 xX $.
    substitution-app.1  $e #Substitution ph2 ph4 ph0 xX $.
    substitution-app    $a #Substitution ( \app ph1 ph2 ) ( \app ph3 ph4 ) ph0 xX $.
         $}
${  $d xX x  $.
    $d y ph0 $.
    substitution-exists.0 $e #Substitution ph2 ph1 y x $.
    substitution-exists.1 $e #Substitution ph3 ph2 ph0 xX $.
    substitution-exists   $a #Substitution ( \exists y ph3 ) ( \exists x ph1 ) ph0 xX
          $. $}
${  $d xX X  $.
    $d Y ph0 $.
    substitution-mu.0 $e #Substitution ph2 ph1 Y X $.
    substitution-mu.1 $e #Substitution ph3 ph2 ph0 xX $.
    substitution-mu   $a #Substitution ( \mu Y ph3 ) ( \mu X ph1 ) ph0 xX $. $}
${  yY-free-in-ph0 $e #Fresh yY ph0 $.
    ph1-definition $e #Substitution ph1 ph0 yY xX $.
    ${  substitution-fold.0   $e #Substitution ph2 ph1 ph3 yY $.
        substitution-fold     $a #Substitution ph2 ph0 ph3 xX $. $}
    ${  substitution-unfold.0 $e #Substitution ph2 ph0 ph3 xX $.
        substitution-unfold   $a #Substitution ph2 ph1 ph3 yY $. $} $}
${  substitution-inverse.0 $e #Fresh xX ph0 $.
    substitution-inverse.1 $e #Substitution ph1 ph0 xX yY $.
    substitution-inverse   $a #Substitution ph0 ph1 yY xX $. $}
${  substitution-fresh.0 $e #Fresh xX ph0 $.
    substitution-fresh   $a #Substitution ph0 ph0 ph1 xX $. $}

$( Application context $)

application-context-var $a #ApplicationContext xX xX $.
${  $d xX ph1 $.
    application-context-app-left.0 $e #ApplicationContext xX ph0 $.
    application-context-app-left   $a #ApplicationContext xX ( \app ph0 ph1 ) $. $}
${  $d 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-substitution.0 $e #Substitution ph0 ph1 ph2 xX $.
    notation-substitution.1 $e #Notation ph3 ph0 $.
    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 #Notation ph1 ph0 $.
    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 ) $. $}

$( Defining not, or, and $)

$c \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-mp.0 $e |- ( \imp ph0 ph1 ) $.
    proof-rule-mp.1 $e |- ph0 $.
    proof-rule-mp   $a |- ph1 $. $}

${  proof-rule-exists.0 $e #Substitution ph0 ph1 y x $.
    proof-rule-exists   $a |- ( \imp ph0 ( \exists x ph1 ) ) $. $}

${  proof-rule-gen.0 $e |- ( \imp ph0 ph1 ) $.
    proof-rule-gen.1 $e #Fresh x ph1 $.
    proof-rule-gen   $a |- ( \imp ( \exists x ph0 ) ph1 ) $. $}
```

```
${  proof-rule-propagation-bot.0 $e #ApplicationContext xX ph0 $.
    proof-rule-propagation-bot.1 $e #Substitution ph1 ph0 \bot xX $.
    proof-rule-propagation-bot   $a |- ( \imp ph1 \bot ) $. $}

${  proof-rule-propagation-or.0 $e #ApplicationContext xX ph0 $.
    proof-rule-propagation-or.1 $e #Substitution ph1 ph0 ( \or ph4 ph5 ) xX $.
    proof-rule-propagation-or.2 $e #Substitution ph2 ph0 ph4 xX $.
    proof-rule-propagation-or.3 $e #Substitution ph3 ph0 ph5 xX $.
    proof-rule-propagation-or   $a |- ( \imp ph1 ( \or ph2 ph3 ) ) $. $}

${  proof-rule-propagation-exists.0 $e #ApplicationContext xX ph0 $.
    proof-rule-propagation-exists.1 $e #Substitution ph1 ph0 ( \exists y ph3 ) xX $.
    proof-rule-propagation-exists.2 $e #Substitution ph2 ph0 ph3 xX $.
    proof-rule-propagation-exists.3 $e #Fresh y ph0 $.
    proof-rule-propagation-exists   $a |- ( \imp ph1 ( \exists y ph2 ) ) $. $}

${  proof-rule-frame.0 $e #ApplicationContext xX ph0 $.
    proof-rule-frame.1 $e #Substitution ph1 ph0 ph3 xX $.
    proof-rule-frame.2 $e #Substitution ph2 ph0 ph4 xX $.
    proof-rule-frame.3 $e |- ( \imp ph3 ph4 ) $.
    proof-rule-frame   $a |- ( \imp ph1 ph2 ) $. $}

${  proof-rule-prefixpoint.0 $e #Substitution ph0 ph1 ( \mu X ph1 ) X $.
    proof-rule-prefixpoint   $a |- ( \imp ph0 ( \mu X ph1 ) ) $. $}

${  proof-rule-kt.0 $e #Substitution ph0 ph1 ph2 X $.
    proof-rule-kt.1 $e |- ( \imp ph0 ph2 ) $.
    proof-rule-kt   $a |- ( \imp ( \mu X ph1 ) ph2 ) $. $}

${  proof-rule-set-var-substitution.0 $e #Substitution ph0 ph1 ph2 X $.
    proof-rule-set-var-substitution.1 $e |- ph1 $.
    proof-rule-set-var-substitution   $a |- ph0 $. $}

proof-rule-existence $a |- ( \exists x x ) $.

${  proof-rule-singleton.0 $e #ApplicationContext xX ph0 $.
    proof-rule-singleton.1 $e #ApplicationContext yY ph1 $.
    proof-rule-singleton.2 $e #Substitution ph3 ph0 ( \and x ph2 ) xX $.
    proof-rule-singleton.3 $e #Substitution ph4 ph1 ( \and x ( \not ph2 ) ) yY $.
    proof-rule-singleton   $a |- ( \not ( \and ph3 ph4 ) ) $. $}
```

B Matching Logic

B.1 Matching Logic Semantics

Here, we define the *semantics* of matching logic. Although semantics is not necessary in generating proof objects, it helps to build the intuition about patterns so we review it here. In a word, matching logic semantics is that of *pattern matching*; that is, a pattern is interpreted as the set of elements that *match* it.

Definition 4. *For a signature Σ, a* (matching logic) Σ-model M *consists of:*

- *a nonempty* carrier set*, which we also denote M;*
- *an* application function $\_\bullet\_ : M \times M \to \mathcal{P}(M)$*, where $\mathcal{P}(M)$ is the powerset;*
- *a* symbol interpretation $\sigma_M \subseteq M$ *as a subset of M, for each $\sigma \in \Sigma$.*

Given a valuation $\rho\colon (EV \cup SV) \to M \cup \mathcal{P}(M)$ *that maps element variables to elements and set variables to sets, i.e., $\rho(x) \in M$ for $x \in EV$ and $\rho(X) \subseteq M$ for $X \in SV$, we inductively define* pattern interpretation $|\varphi|_{M,\rho}$ *as follows:*

1. $|x|_{M,\rho} = \{\rho(x)\}$ *for* $x \in EV$
2. $|X|_{M,\rho} = \rho(X)$ *for* $X \in SV$
3. $|\sigma|_{M,\rho} = \sigma_M$ *for* $\sigma \in \Sigma$
4. $|\varphi_1\, \varphi_2|_{M,\rho} = \bigcup_{a_i \in |\varphi_i|_{M,\rho}, i \in \{1,2\}} a_1 \bullet a_2$*, called* pointwise extension
5. $|\bot|_{M,\rho} = \emptyset$
6. $|\varphi_1 \to \varphi_2|_{M,\rho} = M \setminus (|\varphi_1|_{M,\rho} \setminus |\varphi_2|_{M,\rho})$
7. $|\exists x.\, \varphi|_{M,\rho} = \bigcup_{a \in M} |\varphi|_{M,\rho[a/x]}$
8. $|\mu X.\, \varphi|_{M,\rho} = \mathbf{lfp}(A \mapsto |\varphi|_{M,\rho[A/X]})$

where $\rho[a/x]$ (resp. $\rho[A/X]$) is the updated valuation ρ' with $\rho'(x) = a$ (resp. $\rho'(X) = A$) and agreeing with ρ on any other variables. We use $\mathbf{lfp}(A \mapsto |\varphi|_{M,\rho[A/X]})$ *to denote the least solution A (w.r.t. set inclusion) of $A = |\varphi|_{M,\rho[A/X]}$.*

We compare the above semantics of patterns to that of terms and formulas in FOL. In FOL, terms are interpreted as elements and formulas are interpreted as either true or false. We can restore this classic semantics in matching logic using *functional* and *predicate* patterns, formally defined as follows:

Definition 5. *Let M be a model. We call φ a* functional pattern *or a* term*, if $|\varphi|_\rho$ is a singleton for all ρ, and a* predicate pattern *if $|\varphi|_\rho \in \{\emptyset, M\}$ for all ρ.*

In other words, a (FOL) term t can be regarded as a functional pattern that is matched by exactly one element, which is (represented by) t. A (FOL) formula φ can be regarded as a predicate pattern that is either $\top$ or $\bot$.

Definition 6. *Let Σ be a signature. A* (matching logic) Σ-theory *Γ is a set of Σ-patterns called* axioms*. If a model M satisfies all axioms in Γ, i.e., $|\psi|_{M,\rho} = M$ for all $\psi \in \Gamma$ and valuations ρ, then we write $M \vDash \Gamma$. If $M \vDash \Gamma$ implies $M \vDash \varphi$ for all models M, then we write $\Gamma \vDash \varphi$.*

A matching logic theory Γ plays two roles at a time. On one hand, it restricts the models by enforcing them to satisfy the axioms in Γ. On the other hand, it can be used to derive formal theorems using the matching logic proof system. For proof object generation, we are more interested in deriving formal theorems from Γ, so we introduce matching logic proof system in the following.

B.2 Soundness of Matching Logic Proof System

The soundness of the proof system is established in [5], which we restate below:

Theorem 1 (Soundness Theorem). *$\Gamma \vdash \varphi$ implies $\Gamma \vDash \varphi$ for any Γ and φ.*

B.3 Basic Matching Logic Theories

In this appendix, we give a more detailed account for the theory of equality and sorts aforementioned in Section 5.1.

For convenience we define this abbreviation

$$s_1 \ \ldots \ s_n \ \texttt{\$==} \ s'_1 \ \ldots \ s'_m$$

for the Metamath statements:

```
$c (new constants in s1 ... sn) $.
$a #Pattern s1 ... sn $.
$a #Notation s1 ... sn s'1 ... s'm $.
```

which defines a new construct as a notation using the existing one.

Theory of equality. Recall that to define equality and related notions, we added a *definedness symbol* $\lceil \_ \rceil$ with the axiom:

$$\text{(Definedness)} \quad \lceil x \rceil$$

Based on this, we have the following definitions:

$$\begin{aligned}
&\lfloor \varphi \rfloor \equiv \neg \lceil \neg \varphi \rceil && \text{// Totality (Dual of Definedness)}\\
&\varphi_1 = \varphi_2 \equiv \lfloor \varphi_1 \leftrightarrow \varphi_2 \rfloor && \text{// Equality}\\
&\varphi_1 \subseteq \varphi_2 \equiv \lfloor \varphi_1 \to \varphi_2 \rfloor && \text{// Set Inclusion}\\
&x \in \varphi \equiv \lceil x \wedge \varphi \rceil && \text{// Membership}
\end{aligned}$$

Their semantic meaning is formalized by the following proposition. Note that semantically, $x \subseteq \varphi$ is equivalent to $x \in \varphi$, but we still define the latter notation because it fits well with the intuition that x is an element in φ.

Proposition 1. *Let M be a matching logic Σ-model such that $\lceil \_ \rceil \in \Sigma$ and $M \models$ (Definedness). For any pattern φ, φ', element variable x, and valuation ρ, we have:*

1. $|\lfloor \varphi \rfloor|_{M,\rho} = \begin{cases} M & \text{if } |\varphi|_{M,\rho} = M \\ \emptyset & \text{otherwise} \end{cases}$
2. $|\varphi = \varphi'|_{M,\rho} = \begin{cases} M & \text{if } |\varphi|_{M,\rho} = |\varphi'|_{M,\rho} \\ \emptyset & \text{otherwise} \end{cases}$
3. $|\varphi \subseteq \varphi'|_{M,\rho} = \begin{cases} M & \text{if } |\varphi|_{M,\rho} \subseteq |\varphi'|_{M,\rho} \\ \emptyset & \text{otherwise} \end{cases}$
4. $|x \in \varphi|_{M,\rho} = \begin{cases} M & \text{if } \rho(x) \in |\varphi|_{M,\rho} \\ \emptyset & \text{otherwise} \end{cases}$

We also have the following derivable properties of equality.

Proposition 2. *Suppose the definedness symbol is in the signature. Let Γ be a matching logic theory that contains* (Definedness)*. Then for any patterns $\varphi_1, \varphi_2, \varphi_3$:*

1. $\Gamma \vdash \varphi_1 = \varphi_1$
2. $\Gamma \vdash \varphi_1 = \varphi_2 \to \varphi_2 = \varphi_1$
3. $\Gamma \vdash \varphi_1 = \varphi_2 \wedge \varphi_2 = \varphi_3 \to \varphi_1 = \varphi_3$
4. $\Gamma \vdash \varphi_1 \subseteq \varphi_2 \wedge \varphi_2 \subseteq \varphi_1 \to \varphi_1 = \varphi_2$

In Metamath, we use exactly the same formulation. Namely, we declare the definedness symbol and axiom, and define equality and related notions based on that:

```
$( ⌈_⌉ symbol $)
$c \definedness $.
definedness-is-symbol $a #Symbol \definedness $.
axiom-definedness $a |- ( \app \definedness x ) $.

$( Some notations $)
( \ceil ph0 )         $== ( \app \definedness ph0 )
( \floor ph0 )        $== ( \not ( \ceil ( \not ph0 ) ) )
( \eq ph0 ph1 )       $== ( \floor ( \iff ph0 ph1 ) )
( \included ph0 ph1 ) $== ( \floor ( \imp ph0 ph1 ) )
( \in x ph0 )         $== ( \ceil ( \and x ph0 ) )
```

The results in Proposition 2 can also be formulated and proved in Metamath:

```
lemma-eq-reflexivity  $p |- ( \eq ph0 ph0 ) $= ... $.
lemma-eq-symmetry     $p |- ( \imp ( \eq ph0 ph1 ) ( \eq ph1 ph0 ) ) $= ... $.
lemma-eq-transitivity $p |- ( \imp ( \and ( \eq ph0 ph1 ) ( \eq ph1 ph2 ) )
                                   ( \eq ph0 ph2 ) ) $= ... $.
lemma-eq-intro        $p |- ( \imp ( \and ( \included ph0 ph1 )
                                          ( \included ph1 ph0 ) )
                                   ( \eq ph0 ph1 ) ) $= ... $.
```

Theory of Sorts. Recall from Section 5.1 that we introduced the inhabitant symbol $[\![\_]\!]$ and the sort symbol $Sort$, such that $[\![s]\!]$ can be understood as the inhabitant set of a sort symbol s (or in general a functional pattern φ), and $[\![Sort]\!]$ is the set of all sort symbols.

Our formulation in Metamath is similar:

```
$( Sort symbol $)
$c \sort $.
sort-is-symbol $a #Symbol \sort $.

$( ⟦_⟧ symbol $)
$c \inhabitant $.
inhabitant-is-symbol $a #Symbol \inhabitant $.

$( A few notations $)

( \inh ph0 )               $== ( \app \inhabitant ph0 )
```

```
( \in-sort ph0 ph1 )        $== ( \included ph0 ( \inh ph1 ) )

$( Quantify over the inhabitant(s) of a sort $)
( \sorted-forall x ph0 ph1 ) $== ( \forall x ( \imp ( \in-sort x ph0 ) ph1 ) )
( \sorted-exists x ph0 ph1 ) $== ( \exists x ( \and ( \in-sort x ph0 ) ph1 ) )

$( Quantify over all sort names $)
( \forall-sort x ph0 )      $== ( \sorted-forall x \sort ph0 )
( \exists-sort x ph0 )      $== ( \sorted-exists x \sort ph0 )
```

C Kore and Its Formalization

In this appendix we introduce Kore, which is the bridge between $\mathbb{K}$ and matching logic. We will give a formalization of Kore in Metamath, and then describe our translation procedure from a Kore module to this formalization.

C.1 Syntax of Kore

The following BNF syntax covers a slightly simplified fragment of Kore. We will also omit empty sort annotations (`{ ... }`) whenever possible.

$$
\begin{aligned}
s \in \text{symbol-id} &::= \texttt{\textbackslash\textbackslash?[A-Za-z][A-Za-z0-9\textbackslash\textbackslash-]*} \\
\alpha \in \text{sort-variable} &::= \texttt{[A-Za-z][A-Za-z0-9\textbackslash\textbackslash-]*} \\
x \in \text{element-variable} &::= \texttt{[A-Za-z][A-Za-z0-9\textbackslash\textbackslash-]*}
\end{aligned}
$$

$$
\begin{aligned}
\theta \in \text{sort} &::= s\ \texttt{\{}\ \theta^*\ \texttt{\}} \mid \alpha \\
\varphi \in \text{pattern} &::= x\ \texttt{:}\ \theta \\
&\mid n \in \mathbb{Z} \mid b \in \{\texttt{true}, \texttt{false}\} \\
&\mid \texttt{\textbackslash top \{}\ \theta\ \texttt{\} ( )} \\
&\mid \texttt{\textbackslash bottom \{}\ \theta\ \texttt{\} ( )} \\
&\mid \texttt{\textbackslash not \{}\ \theta\ \texttt{\} (}\ \varphi\ \texttt{)} \\
&\mid \texttt{\textbackslash and \{}\ \theta\ \texttt{\} (}\ \varphi_1\ \texttt{,}\ \varphi_2\ \texttt{)} \\
&\mid \texttt{\textbackslash or \{}\ \theta\ \texttt{\} (}\ \varphi_1\ \texttt{,}\ \varphi_2\ \texttt{)} \\
&\mid \texttt{\textbackslash exists \{}\ \theta_2\ \texttt{\} (}\ x\ \texttt{:}\ \theta_1\ \texttt{,}\ \varphi\ \texttt{)} \\
&\mid \texttt{\textbackslash forall \{}\ \theta_2\ \texttt{\} (}\ x\ \texttt{:}\ \theta_1\ \texttt{,}\ \varphi\ \texttt{)} \\
&\mid \texttt{\textbackslash equals \{}\ \theta_1\ \texttt{,}\ \theta_2\ \texttt{\} (}\ \varphi_1\ \texttt{,}\ \varphi_2\ \texttt{)} \\
&\mid \texttt{\textbackslash next \{}\ \theta\ \texttt{\} (}\ \varphi\ \texttt{)} \\
&\mid \texttt{\textbackslash rewrites \{}\ \theta\ \texttt{\} (}\ \varphi_1\ \texttt{,}\ \varphi_2\ \texttt{)} \\
\text{sentence} &::= \texttt{sort}\ s\ \texttt{\{}\ \alpha^*\ \texttt{\}} \\
&\mid \texttt{symbol}\ s\ \texttt{\{}\ \alpha^*\ \texttt{\} (}\ \theta^*\ \texttt{) :}\ \theta'
\end{aligned}
$$

$$
\begin{aligned}
& \mid \texttt{axiom \{ } \alpha^* \texttt{ \}}\ \varphi \\
\text{module} \ ::= {} & \texttt{module}\ s\ \text{sentence}^*\ \texttt{endmodule}
\end{aligned}
$$

Each Kore module specifies a theory in a many-sorted version of matching logic, and each Kore pattern is essentially a sorted matching logic pattern. A more detailed formulation of many-sorted matching logic can be found in [6].

Ideally, all features of $\mathbb{K}$ can be described in Kore. For instance, the following fragment in Figure 3

```
syntax State ::= "<" Int "," Int ">"
rule <M, N> => <M -Int 1, N +Int M>
     requires M >Int 0
```

can be translated to these equivalent sentences in Kore:

```
sort State
symbol \state(Int, Int): State

// \state(M:Int, N:Int) is a functional pattern
axiom {R}
    \forall{R}(M:Int, \forall{R}(N:Int,
    \exists{R}(S:State, \equals{State, R}(S:State, \state(M:Int, N:Int)))))

// Omitting axioms saying that \state is a constructor

// Rewriting axiom
axiom \rewrites{State}(
    \and{State}(\state(M:Int, N:Int),
                \equals{Bool, State}(>Int(M:Int, N:Int), true)),
    \state(-Int(M:Int, 1), +Int(N:Int, M:Int))
)
```

C.2 Formalizing Kore in Matching Logic

In this section, we formally define the semantics of Kore as a theory in matching logic. We will use the definitions of equality and sort-related constructs, as well as the notation `$==`, introduced in Appendix B.3.

We will start with the semantics of the propositional fragment of Kore:

```
( \kore-bottom ph0 )      $== \bot
( \kore-top ph0 )         $== ( \inh ph0 )
( \kore-not ph0 ph1 )     $== ( \and ( \not ph1 ) ( \inh ph0 ) )
( \kore-and ph0 ph1 ph2 ) $== ( \and ph1 ph2 )
( \kore-or ph0 ph1 ph2 )  $== ( \or ph1 ph2 )

( \kore-implies ph0 ph1 ph2 )
                          $== ( \kore-or ph0 ( \kore-not ph0 ph1 ) ph2 )
( \kore-iff ph0 ph1 ph2 ) $== ( \kore-and ph0 ( \kore-implies ph0 ph1 ph2 )
                                             ( \kore-implies ph0 ph2 ph1 ) )
```

These notations corresponds directly to the actual syntax of Kore in Appendix C.1. For example `\and { θ } ( φ1 , φ2 )` can be directly translated to `( \kore-and θ φ1 φ2 )`. Note that although being ignored in most cases, the sort annotation still plays an important part in, for example, `\kore-not`.

Sorted quantifiers are defined as follows:

```
( \kore-exists ph0 ph1 x ph2 ) $== ( \and ( \sorted-exists x ph0 ph2 )
                                          ( \inh ph1 ) )

( \kore-forall-sort x ph0 )    $== ( \forall-sort x ph0 )

$( Dual of \kore-exists $)
( \kore-forall ph0 ph1 x ph2 ) $== ( \kore-not ph1
                                   ( \kore-exists ph0 ph1 x
                                   ( \kore-not ph1 ph2 ) ) )
```

Once we have defined these basic Kore patterns, the validity of a pattern can be defined as:

```
( \kore-valid ph0 ph1 ) $== ( \eq ph1 ( \inh ph0 ) )
```

That is, a Kore pattern of sort `ph0` is valid iff its interpretation is exactly the inhabitant set of the sort `ph0`. This coincides with the definition of validity of a many-sorted matching pattern [5].

Now we can introduce the theory of equality using the definitions in Appendix B.3. We first define the Kore counterpart of $\lceil \_ \rceil$:

```
( \kore-ceil ph0 ph1 ph2 ) $== ( \and ( \ceil ph2 ) ( \kore-top ph1 ) )
```

Then $\lfloor \_ \rfloor$ and equality can be defined as follows:

```
( \kore-floor ph0 ph1 ph2 )      $== ( \kore-not ph1
                                         ( \kore-ceil ph0 ph1
                                         ( \kore-not ph0 ph2 ) ) )

( \kore-equals ph0 ph1 ph2 ph3 ) $== ( \kore-floor ph0 ph1
                                         ( \kore-iff ph0 ph2 ph3 ) )
```

Informally, the semantics of `\kore-ceil` is that given a pattern `ph2` of sort `ph0`, `( \kore-ceil ph0 ph1 ph2 )` is the entire inhabitant of `ph1` (i.e. it's valid) if `ph2` is not empty; otherwise `( \kore-ceil ph0 ph1 ph2 )` is falsum. The reason why two sorts are involed is because when using `\kore-equals`, the context where such pattern is used may expect a different sort than the patterns being compared in `\kore-equals`. For example, if we want to describe the set of concrete pairs of integers such that the first is greater than the second, we may use the following Kore pattern:

```
// Definition of Pair, \pair, Bool, Int, >Int are omitted
\and{Pair}(\pair(M:Int, N:Int), \equals{Bool, Pair}(>Int(M:Int, N:Int), true))
```

Here, `\equals{Bool, Pair}` needs to embed the truth value of the equality into the inhabitant set of `Pair` in order for it to be well-sorted.

Finally, we can define the theory of rewriting:

```
$c \kore-next-symbol $.
kore-next-is-symbol $a #Symbol \kore-next-symbol $.

$( Any interpretation of \kore-next must be well-sorted $)
kore-next-sorting $a |- ( \imp ( \in-sort ph1 ph0 )
                               ( \in-sort ( \kore-next ph0 ph1 ) ph0 ) ) $.

( \kore-next ph0 ph1 )         $== ( \app \kore-next-symbol ph1 )
( \kore-rewrites ph0 ph1 ph2 ) $== ( \kore-implies ph0 ph1
                                       ( \kore-next ph0 ph2 ) )
```

Note that `\kore-next` corresponds to the *one-path next* symbol • introduced in Section 5.1. We can also define the eventually pattern and reflexive transitive closure of `\kore-rewrites`, in the same manner as in Section 5.1:

```
${
    $d X ph0 $.
    $d X ph1 $.
    ( \kore-eventually ph0 ph1 )
        $== ( \mu X ( \kore-or ph0 ph1 ( \kore-next ph0 X ) ) )
$}

$( Reflexive transitive closure of \kore-rewrites $)
( \kore-rewrites-star ph0 ph1 ph2 )
    $== ( \kore-implies ph0 ph1 ( \kore-eventually ph0 ph2 ) )
```

It can then be shown in Metamath that `\kore-rewrites-star` is reflexive and transitive:

```
${
    $( Assume that ph1 is well-sorted $)
    kore-rewrites-star-reflexivity.0 $e |- ( \in-sort ph1 ph0 ) $.
    kore-rewrites-star-reflexivity
        $p |- ( \kore-valid ph0 ( \kore-rewrites-star ph0 ph1 ph1 ) ) $= ... $.
$}

${
    kore-rewrites-star-transitivity.0 $e |- ( \in-sort ph1 ph0 ) $.
    kore-rewrites-star-transitivity.1 $e |- ( \in-sort ph2 ph0 ) $.
    kore-rewrites-star-transitivity.2 $e |- ( \in-sort ph3 ph0 ) $.

    kore-rewrites-star-transitivity.3
        $e |- ( \kore-valid ph0 ( \kore-rewrites-star ph0 ph1 ph2 ) ) $.
    kore-rewrites-star-transitivity.4
        $e |- ( \kore-valid ph0 ( \kore-rewrites-star ph0 ph2 ph3 ) ) $.

    kore-rewrites-star-transitivity
        $p |- ( \kore-valid ph0 ( \kore-rewrites-star ph0 ph1 ph3 ) ) $= ... $.
$}
```

C.3 Translating Kore to Metamath

As mentioned in Section 5.2, we have a two-step translation from $\mathbb{K}$ to Kore, and from Kore to Metamath. In this section we describe the latter translation procedure from Kore to its formalization in Metamath.

Translating Kore Patterns. We start from translating a single Kore pattern or sort. For each Kore pattern/sort φ, we will emit a Metamath term as the encoding of the pattern/sort, denoted $[\varphi]$, potentially also adding a few auxillary constant and variable statements to define new variables or domain values. The encoding procedure is defined inductively as follows:

- For an element variable $x : \theta$ or a sort variable α, we encode it as a Metamath variable with a unique name.
- For a pattern/sort of the form `s { θ1, ..., θm } ( φ1, ..., φn )` ($m \geq 0, n \geq 0$), we encode it as

  $$\texttt{(}\ s'\ [\theta_1]\ \ldots\ [\theta_m]\ [\varphi_1]\ \ldots\ [\varphi_n]\ \texttt{)}.$$

  if $m + n \neq 0$, otherwise we simply emit s'. The correspondence between s and s' is shown in the following table:

s	s'
`\top`	`\kore-top`
`\bottom`	`\kore-bottom`
`\not`	`\kore-not`
`\and`	`\kore-and`
`\or`	`\kore-or`
`\equals`	`\kore-equals`
`\next`	`\kore-next`
`\rewrites`	`\kore-rewrites`

  We will also allow s to be any user-defined sort or symbol, in which case s' would be a suitable encoding of the identifier of the sort/symbol.
- For a quantifier pattern of the form `s { θ1 } ( x : θ2 , φ )`, where $s \in$ { `\exists`, `\forall` }, we encode it as

  $$\texttt{(}\ s'\ [\theta_2]\ [\theta_1]\ [x]\ [\varphi]\ \texttt{)}$$

  where s' = `\kore-exists` if s = `\exists`, otherwise s' = `\kore-forall` if s = `\forall`.

For domain values $n \in \mathbb{Z}$ or $b \in$ { `true`, `false` }, since we currently do not have an axiomatization of natural numbers or booleans in matching logic in Metamath, we leave them as uninterpreted matching logic symbols.

Translating Kore Sentences. We now consider the three types of sentences in Kore whose syntax is defined in Appendix C.1. Each sentence will be translated to one or more Metamath statements.

First, we add a new notation for convenience:

```
${
    $d x ph0 $.
    ( \kore-is-sort ph0 ) $== ( \exists-sort x ( \eq x ph0 ) )
$}
```

That is, `ph0` is a functional pattern in set of sort names.

In the following, we will use `( \app` t_1 `...` t_n `)` and `( \and` t_1 `...` t_n `)` as abbreviations for `( \app ( \app` t_1 t_2 `) ...` t_n `)` and `( \and ( \and` t_1 t_2 `) ...` t_n `)`, respectively. $[\varphi]$, as defined before, denotes the encoding of a pattern or sort φ. We will use `ph`$_i$ ($i \in \mathbb{N}$) to denote a pattern metavariable in Metamath, that is, a Metamath variable `ph`$_i$ declared in the following way:

```
$v phᵢ $.
$f #Pattern phᵢ $.
```

We encode the three types of sentences as follows:

– For a sort definition of the form

$$\texttt{sort}\ s\ \texttt{\{}\ \alpha_1, \ldots, \alpha_n\ \texttt{\}}$$

We add the Metamath statements

```
$c [s] $.
$a #Symbol [s] $.
( [s] ph1 ... phn ) $== ( \app [s] ph1 ... phn )

$a |- ( \imp ( \and ( \kore-is-sort ph1 ) ... ( \kore-is-sort phn ) )
            ( \kore-is-sort ( [s] ph1 ... phn ) ) ) $.
```

– For a symbol definition of the form

$$\texttt{symbol}\ s\ \texttt{\{}\ \alpha_1, \ldots, \alpha_m\ \texttt{\}}\ \texttt{(}\ \theta_1, \ldots, \theta_n\ \texttt{)}\ \texttt{:}\ \theta'$$

We add the Metamath statements:

```
$c [s] $.
$a #Symbol [s] $.
( [s] ph1 ... ph(m+n) ) $== ( \app [s] ph1 ... ph(m+n) )

$( Sorting axiom $)
$a |- ( \imp ( \and ( \kore-is-sort [α1] ) ... ( \kore-is-sort [αm] ) )
      ( \imp ( \and ( \in-sort ph(m+1) [θ1] ) ... ( \in-sort ph(m+n) [θn] ) )
            ( \in-sort ( [s] ph1 ... ph(m+n) ) ) ) $.
```

– For an axiom of the form

$$\texttt{axiom}\ \texttt{\{}\ \alpha_1, \ldots, \alpha_n\ \texttt{\}}\ \varphi$$

We add the Metamath statements:

```
$a |- ( \kore-forall-sort α₁ ... ( \kore-forall-sort αₙ
      ( \kore-valid [θ] [φ] ) ) ... ) $.
```

where we infer the sort θ syntactically from φ.

D An Interactive Theorem Prover for Matching Logic

In an engineering effort to simplify our work in proving basic theorems in matching logic, we developed an *interactive theorem prover* (ITP) for matching logic. This allows us to automate procedures for proving some simple theorems. For example, as discussed in Section 4.2, our formulation of matching logic in Metamath use a metalevel relation `#Notation` to define new notations. Proving a `#Notation` claim is straightforward using the axioms of `#Notation`, but it can be a tedious task. Using the ITP, on the other hand, one could prove such theorem using an *automated tactic*.

In the following sections, we briefly describe the usage of ITP and some features.

D.1 Basic Usage.

The current version of the ITP supports what can be understood as *backward proofs*. One would start with a Metamath statement to be proved, which we will refer to as a *goal*, and use a sequence of *tactics* to resolve a goal or reduce it to potentially simpler goal(s), until all goals are resolved.

For instance, to prove the following theorem in Metamath (see Section 4.2 for our formulation of matching logic in Metamath):

```
${
    not-elim.0 $e |- ph0         $.
    not-elim.1 $e |- ( \not ph0 ) $.
    not-elim   $p |- \bot        $= ? $.
$}
```

This is essentially just an instance of modus ponens since `( \not ph0 )` is a notation for `( \imp ph0 \bot )`. We can load it into the ITP, which would output (we will use `$? ... $.` to denote a goal):

```
====================
essential(s):
  not-elim.0 $e |- ph0 $.
  not-elim.1 $e |- ( \not ph0 ) $.
====================
goal(s):
  $? |- \bot $.
>
```

Then we can apply modus ponens using the tactic:

```
> apply proof-rule-mp
goal(s):
  $? |- ( \imp $0 \bot ) $.
  $? |- $0 $.
```

`$0` is a temporary placeholder for any Metamath term (i.e. a metavariable that is left unassigned after unification). In this case we can explicitly assign it with `ph0` using the tactic `let`.

```
> let $0 = "ph0"
goal(s):
  $? |- ( \imp ph0 \bot ) $.
  $? |- ph0 $.
```

Then the rest is straightforward: one needs to show that `|- ( \not ph0 )` is just a notation for `|- ( \imp ph0 \bot )`, and the second goal is by hypothesis. To do that, we first apply the axiom `notation-proof`, which is:

```
${
    notation-proof.0 $e |- ph0 $.
    notation-proof.1 $e #Notation ph1 ph0 $.
    notation-proof   $a |- ph1 $.
$}
```

This reduces the first goal to two more goals:

```
> apply notation-proof
goal(s):
  $? |- $2 $.
  $? #Notation ( \imp ph0 \bot ) $2 $.
  $? |- ph0 $.
```

We assign `$2` with `( \not ph0 )`:

```
> let $2 = "( \not ph0 )"
goal(s):
  $? |- ( \not ph0 ) $.
  $? #Notation ( \imp ph0 \bot ) ( \not ph0 ) $.
  $? |- ph0 $.
```

Once we resolve the first and third rule by applying the hypotheses `not-elim.0` and `not-elim.1`, we can resolve the second rule using an automated tactic called `notation`.

```
> apply not-elim.1
...
> notation
...
> apply not-elim.0
====================
no goals left!
```

The complete proof script would be

```
apply proof-rule-mp
let $0 = "ph0"
apply notation-proof
let $2 = "( \not ph0 )"
apply not-elim.1
notation
apply not-elim.0
```

Finally, a Metamath-checkable proof can be generated using the `proof` command:

```
> proof
ph0-is-pattern bot-is-pattern ph0-is-pattern not-is-pattern ph0-is-pattern
bot-is-pattern imp-is-pattern not-elim.1 ph0-is-pattern not-is-pattern
ph0-is-pattern bot-is-pattern imp-is-pattern ph0-is-pattern not-is-pattern
ph0-is-pattern bot-is-pattern imp-is-pattern ph0-is-pattern bot-is-pattern
imp-is-pattern ph0-is-pattern not-is-sugar ph0-is-pattern bot-is-pattern
imp-is-pattern ph0-is-pattern bot-is-pattern imp-is-pattern ph0-is-pattern
bot-is-pattern imp-is-pattern notation-reflexivity notation-symmetry
notation-transitivity notation-symmetry notation-proof not-elim.0 proof-rule-mp
```

D.2 Desugaring Notations.

As shown in the previous example, the `notation` tactic can be used to prove simple `#Notation` statements:

```
goal(s):
  $? #Notation ( \forall x ph0 ) ( \not ( \exists x ( \not ph0 ) ) ) $.
> notation
no goals left!
```

Similarly, one can also use the `desugar` tactic to directly expand notations in a goal. For instance:

```
goal(s):
  $? |- ( \imp ( \forall x ph0 ) ph0 ) $.
> desugar \forall
goal(s):
  $? |- ( \imp ( \not ( \exists x ( \not ph0 ) ) ) ph0 ) $.
```

In the implementation, we use a naive algorithm to normalize patterns to their canonical form, then use the axioms of `#Notation` to compose each step of normalization. As a future improvement, we can search for a shortest path of notation expansion between two patterns, which can reduce the size of the resulting proof.

D.3 Automatic Propositional Proofs.

Although propositional tautologies pose no theoretical difficuty, proving them in practice can still be a considerable amount of work. For instance, the well-known formulation of ZFC set theory in Metamath contains over 1000 manually-proved propositional theorems [19]. Since we are not primarily aiming at minimizing the proof length, we added an automated tactic called `tautology` for proving any propositional fact, which has significantly reduced our work.

The `tautology` tactic uses a refutation theorem proving method by utilizing the resolution rule:

$$\frac{\varphi_1 \vee \dots \varphi_m \vee \psi \qquad \varphi_1' \vee \dots \varphi_n' \vee \neg\psi}{\varphi_1 \vee \dots \varphi_m \vee \varphi_1' \vee \dots \vee \varphi_n'}$$

where all formulae are literals (atomic propositions or negation of atomic propositions). The resolution rule is *refutation complete* [22], in the sense that for a set of propositional formulas Σ clauses (disjunction of literals), if $\Sigma \models \bot$ in the standard model, then $\Sigma \vdash \bot$ using only the refutation rule.

We have manually proved a number of facts (including the resolution rule itself) in Metamath that can be used to normalize every propositional pattern in matching logic to a CNF. Then the tactic will take a valid propositional pattern φ in matching logic, (provably) transform $\varphi \to \bot$ to CNF, and use the resolution rule to prove $(\varphi \to \bot) \to \bot$, which implies φ.