Technical Report: An Interactive Theorem Prover for Matching Logic with Proof Object Generation

Zhengyao Lin
zl38@illinois.edu

Xiaohong Chen
xc3@illinois.edu

Grigore Roşu
grosu@illinois.edu

University of Illinois at Urbana-Champaign

October 5, 2021

Abstract

Matching logic is a uniform logical foundation for $\mathbb{K}$, which is a language semantics framework with the philosophy that all tooling around a language should be automatically generated from a single, rigorous definition of its formal semantics. In practice, $\mathbb{K}$ has been widely used to define the formal semantics of many real-world languages and to generate their execution and verification tools. However, there lacks a generic theorem prover that connects $\mathbb{K}$ with its logical foundation—matching logic. In this paper, we present ITP_{ML}, which is the first interactive theorem prover for matching logic. The main advantage of ITP_{ML} is its ability to generate machine-checkable proof objects as certificates that witness the correctness of its formal reasoning. ITP_{ML} is built on top of Metamath, a language to define formal systems, which allows it to have a small trust base of only 250 lines of code. ITP_{ML} supports both backward and forward proofs, allows users to dynamically add intermediate lemmas, and features automated proof tactics for common utilities such as reasoning about notations and proving propositional tautologies.

1 Introduction

The $\mathbb{K}$ framework (`https://kframework.org`) is a language semantics framework that allows one to define the formal operational semantics of a programming language, from which various language tools, such as parsers, interpreters, deductive verifiers, symbolic execution engines, and model checkers, are automatically generated by $\mathbb{K}$. $\mathbb{K}$ has been successfully applied to define the formal semantics of many real-world languages such as C [10], EVM [13], x86-64 [8], Java [2], JavaScript [22], and Python [11].

In addition to its practical application, there is active research that studies the theoretical aspects of $\mathbb{K}$. In [25, 6], the authors propose *matching logic* as the logical foundation of $\mathbb{K}$, in the sense that (1) all language definitions in $\mathbb{K}$ become matching logic axioms and theories; and (2) all language tools generated by $\mathbb{K}$ can be specified using matching logic formulas and their correctness can be witnessed by matching logic proofs. Therefore, matching logic is a uniform logic to specify and reason about programs and their dynamic properties in $\mathbb{K}$.

Unfortunately, the current implementation of $\mathbb{K}$ has not been fully verified using matching logic, due to its complexity. In our recent work [4, 16], we take a step towards filling this gap by generating machine-checkable proof objects to certify the correctness of program execution and verification in $\mathbb{K}$.

This motivates the development of an interactive theorem prover, since we need to formalize matching logic and $\mathbb{K}$ in a system with a small trust base and, at the same time, enough automation for proving important matching logic theorems on which the correctness of $\mathbb{K}$ relies. Existing mature interactive theorem provers such as Coq [18] and Isabelle [28] are more than adequate to formalize matching logic, but they both have large trust bases. Therefore, we turn to Metamath [20], which is a minimalistic language to define formal systems, and we build our interactive theorem prover on top of Metamath by providing higher level reasoning tactics for matching logic, while generating proof objects in Metamath to significantly reduce our trust base.

Thus, the **main contribution** of this paper is an interactive theorem prover ITP_{ML} that makes it possible to carry out arbitrary matching logic reasoning and generate machine-checkable proof objects as correctness certificates. As the first interactive theorem prover for matching logic, ITP_{ML} has greatly helped in the research work [4, 16], in which more than 600 matching logic theorems are proved using ITP_{ML}. ITP_{ML} supports proof tactics for:

- Backward and forward reasoning (Section 3.2),
- Dynamically adding claims during a proof (Section 3.2),
- Transparent handling of user-defined notations (Section 3.4), and
- Automated reasoning of propositional tautologies (Section 3.5).

For all these features, one does not need to trust the implementation of ITP_{ML}, and the only trust base is a 250-line formalization/encoding of matching logic in Metamath [14].

The rest of the paper is organized as follows. In Section 2, we introduce matching logic and its formalization/encoding in Metamath, which is the basis of our ITP_{ML}. In Section 3, we present ITP_{ML} in detail. We discuss its main proof tactics and explain how to generate proof objects for the tactics. In Section 4, we conclude the paper with related work and future directions.

2 Matching Logic

In this section we give a brief introduction to matching logic [26, 25, 6] which our interactive theorem prover ITP_{ML} is based on. As a logic, matching logic can be seen as an extension to both first-order logic (FOL) and modal μ-calculus [15]:

- It extends FOL with fixpoint operators and interprets each term/formula uniformly as a set of elements that match it. As such, there is no distinction between terms and formulas, which we uniformly call *patterns* (Definition 1).
- It can also be seen as a first-order extension of the modal μ-calculus, allowing us to express temporal properties which are useful in program verification.

These two components collectively give rise to a logic that is suitable for reasoning about programs. The FOL fragment allows us to describe complex properties statically, while the modal μ-calculus fragment allows us to reason about dynamic/temporal properties for program execution and verification. Matching logic is therefore developed as a logical foundation of the $\mathbb{K}$ framework.

In Section 2.1, we review the syntax of matching logic. In Section 2.2, we show the Hilbert-style proof system of matching logic and our formalization of it in Metamath, which our ITP_{ML} is based on.

2.1 Matching Logic Syntax

We introduce the syntax of matching logic, following [25, 6, 5]. We fix two disjoint sets of variable $\mathbb{EV}$ and $\mathbb{SV}$. We use $x, y, \ldots$ to range over $\mathbb{EV}$, called the *element variables*; and we use $X, Y, \ldots$ to range over $\mathbb{SV}$, called the *set variables*. The *syntax* of matching logic is parametric in a *signature* Σ, whose elements are called *symbols* denoted σ. Matching logic formulas, called *patterns*, are formally defined as follows:

Definition 1. For a signature Σ, a *(matching logic)* Σ*-pattern* φ is inductively defined by the following grammar:

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

where in $\mu X.\, \varphi$ we require φ to be *positive* in X, that is, the occurrences of X in φ are nested in the left-hand side of implications in an even number of times.

Here the pattern $\varphi_1\, \varphi_2$ denotes the *application*, where φ_1 is applied to φ_2. So matching logic is an unsorted logic whose fomulas (patterns) are built from variables, symbols, FOL constructs $\bot$, $\to$, $\exists$, a binary application operator, and a least fixpoint construct μ.

As usual, we use $\mathrm{FV}(\varphi) \subseteq \mathbb{EV} \cup \mathbb{SV}$ to denote the set of free variables in φ, and $\varphi[\psi/x]$ and $\varphi[\psi/X]$ to denote capture-free substitution, where α-renaming happens implicitly to prevent variable capture. We define the following common constructs as derived constructs, i.e., notations, in the usual way:

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

Informally, the *semantics* of matching logic interprets each pattern as a set of element that *matches* it, and the logical connectives have semantics similar to set operations. For example, if we fix some domain M, if some element $a \in M$ matches patterns φ and ψ, then a matches $\varphi \wedge \psi$; if a matches $\varphi[b/x]$ for all $b \in M$, then a matches $\forall x.\varphi$. That is, $\wedge$ is interpreted as binary set intersection and $\forall$ is interpreted as intersection over the entire domain.

Definition 2. Let Σ be a signature. A *(matching logic)* Σ*-theory* Γ is a set of Σ-patterns called *axioms*.

As discussed in the introduction of this section, matching logic is the logical foundation of $\mathbb{K}$. More concretely, a $\mathbb{K}$ definition of a programming language L is translated into a signature Σ^L and a matching logic theory Γ^L. Γ^L would contain axioms that describe both the syntax and semantics of the language L, which allow us to formally reason about statements involving properties of the language or a specific program.

2.2 Matching Logic Proof System and its Formalization

Matching logic has a *Hilbert-style* proof system, of which we show two important fragments in Figure 1 and refer reader to [6] for the full system. The two fragments shown in Figure 1 are FOL reasoning rules providing complete FOL reasoning and fixpoint reasoning similar to that of modal μ-calculus [15].

Consequently, we have the following notion of provability:

Definition 3. We say φ *is provable from* Σ, denoted $\Sigma \vdash \varphi$, if there exists a Hilbert-style proof of φ using the axioms and inference rules in the proof system of [6], and formulas in Σ.

2.2.1 A Formalization in Metamath.

In previous work [4], we formalized the syntax and proof system of matching logic as presented above. Our formalization is encoded in Metamath [20], which is a minimalistic language to specify formal systems and proofs in them in a machine-checkable fashion. The most influential use of Metamath is the formalization

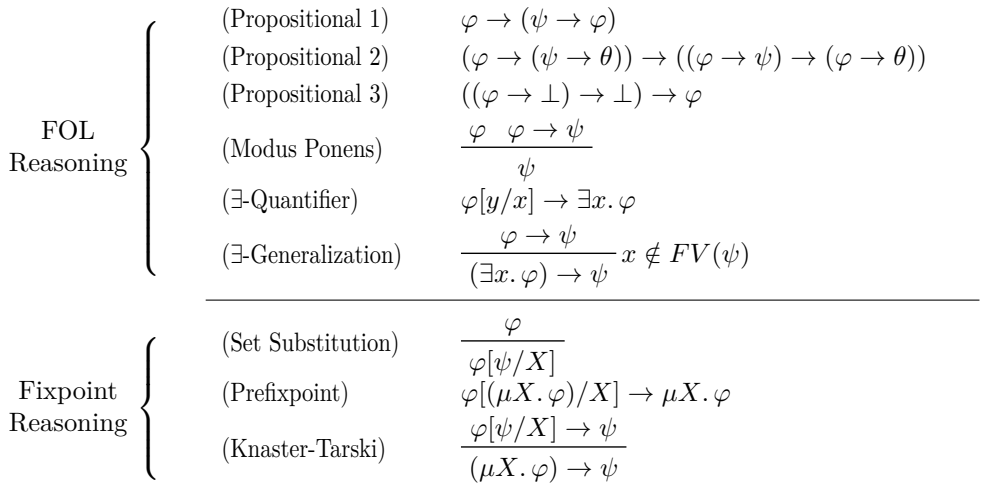

Figure 1: Selected rules from a matching logic proof system.

of the ZFC set theory [19], from which more than 23,000 complete mathematical proofs have been worked out in full detail.

Our interactive theorem prover ITP_{ML} is built on top of this formalization as a tool to help generating proofs in Metamath format. Therefore, we introduce basic notations in Metamath and give an overview of our matching logic formalization.

In essence, Metamath provides bare-bones mechanisms for a user to define a formal system using axioms and inference rules. This usually starts with defining the *well-formedness* of its formulas/term, and then a proof system.

For instance, in the case of matching logic [4], the (Modus Ponens) rule is

```
${ mp.0 $e |- ( \imp ph0 ph1 ) $.
   mp.1 $e |- ph0 $.
   mp   $a |- ph1 $. $}
```

which in common notation corresponds to the inference rule

$$\frac{\texttt{|- (\imp } \varphi_0\ \varphi_1\ \texttt{)} \qquad \texttt{|- } \varphi_0}{\texttt{|- } \varphi_1}\ (\texttt{mp})$$

Here we use `|-` to denote the provability relation in the proof system in Figure 1, and `\imp` to denote implication. The first two *statements* starting with `mp.0` and `mp.1` are *essential statements* (`$e`) that act as hypotheses, and the third *axiomatic statement* (`$a`) states an axiom with the given hypotheses. The parentheses `${` and `$}` indicate the scope of the hypotheses.

Once the axioms are declared, one can write down a *provable statement* (which can also have hypotheses). For example:

```
imp-refl $p |- ( \imp ph0 ph0 ) $= $( proof omitted $) $.
```

which says that $\varphi_0 \to \varphi_0$ is provable for any pattern φ_0. A *proof* of this statement would simply be an encoding of a proof tree with the conclusion $\vdash \varphi_0 \to \varphi_0$, using any previously declared axioms and/or provable statements.

Besides `|-`, we also defined other *metalevel* relations in our formalization in order to describe the proof system. One of the important relations is `#Notation`, which axiomatizes the usual concept of denoting one logical connective using others. `#Notation` is an equivalence relation, that is,

```
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 $. $}
```

It is also congruent with respect to the syntax of patterns and other metalevel relations. To rephrase some of the notations defined in Section 2.1, we have

```
not $a #Notation ( \not ph0 )     ( \imp ph0 \bot ) $. $(  \bot denotes ⊥  $)
or  $a #Notation ( \or ph0 ph1 )  ( \imp ( \not ph0 ) ph1 ) $.
and $a #Notation ( \and ph0 ph1 ) ( \not ( \or ( \not ph0 ) ( \not ph1 ) ) ) $.
```

One problem now is that even though these axioms make sense, using them in practice would still require us to go through all the definitions and axioms to produce a "proof for notations". This is where our interactive theorem prover ITP_{ML} comes in. ITP_{ML} is equipped with proof automation specialized to our formalization of matching logic, and in ITP_{ML}, notations are transparent to users (Section 3.4). These small pieces of proof automation allow us to focus on higher level ideas of a proof.

3 Interactive Theorem Prover for Matching Logic: ITP_{ML}

In this section, we introduce the design and implementation of our interactive theorem prover ITP_{ML}. ITP_{ML} allows users to carry out arbitrary matching logic reasoning within any (user-provided) axioms and/or theories. A user can define any matching logic theory in ITP_{ML}, from which theorems can be proved. ITP_{ML} provides a human-readable tactic language (see Section 3.2) with proof tactics that support both backward and forward proof styles.

ITP_{ML} also provides automated proof tactics that are specific to matching logic and its formalization, such as those for reasoning about notations and propositional tautologies. All tactics are trustworthy because ITP_{ML} generate proof objects for them that are directly checkable by the Metamath proof checkers (see Section 3.3). Therefore, our ITP_{ML} has a very small trust base, namely the matching logic formalization itself (about 250 LOC in Metamath) [14].

In the following, we first give an example of using ITP_{ML} in Section 3.1. We define the proof state and basic tactics in more detail in Section 3.2. Then we describe how ITP_{ML} generates a Metamath-compatible proof in Section 3.3. Finally, we discuss our automated tactics for notations and propositional tautologies in Section 3.4 and Section 3.5.

3.1 An Example: Propositional Reasoning

As a simple example, we use ITP_{ML} to show how to prove that "if-and-only-if" $\leftrightarrow$ is symmetric, that is,

```
${ iff-sym.0 $e |- ( \iff ph0 ph1 ) $.
   iff-sym   $p |- ( \iff ph1 ph0 ) $= ? $. $}
```

where `?` is a placeholder for the proof. Although being simple, it illustrates some core features of ITP_{ML}: backward reasoning and forward reasoning using `apply` and `from` tactics (Section 3.2), and automated reasoning of notations (Section 3.4). More complex proofs and proof tactics work in a similar way.

We load this theorem into ITP_{ML}. The initial *proof state* is shown below:

```
hypotheses:
  iff-sym.0 $e |- ( \iff ph0 ph1 ) $.
goal(s):
  $? |- ( \iff ph1 ph0 ) $.
```

```
> $( prompt to input proof tactic $)
```

which says that we are given a hypothesis `|- ( \iff ph0 ph1 )`, and our current *goal* is `|- ( \iff ph1 ph0 )`.

To ease our presentation, let us assume that the following theorems have been proved and added to the underlying theory. These theorems are essentially the natural deduction rules for $\wedge$.

```
${ and-intro.0      $e |- ph0 $.
   and-intro.1      $e |- ph1 $.
   and-intro        $p |- ( \and ph0 ph1 ) $= $( proof omitted $) $. $}

${ and-elim-left.0  $e |- ( \and ph0 ph1 ) $.
   and-elim-left    $p |- ph0 $= $( proof omitted $) $. $}

${ and-elim-right.0 $e |- ( \and ph0 ph1 ) $.
   and-elim-right   $p |- ph1 $= $( proof omitted $) $. $}
```

Given the above theorems, we can prove the goal as follows. First, recall that $\leftrightarrow$ is defined as a notation:

```
$a #Notation ( \iff ph0 ph1 ) ( \and ( \imp ph0 ph1 ) ( \imp ph1 ph0 ) ) $.
```

Since ITP_{ML} unifies patterns modulo the relation defined by `#Notation` (see Section 3.4), we can directly apply `and-intro` to our proof goal:

```
goal(s):
  $? |- ( \iff ph1 ph0 ) $.

> apply and-intro

goal(s):
  $? |- ( \imp ph1 ph0 ) $.
  $? |- ( \imp ph0 ph1 ) $.
>
```

Then to eliminate the conjunction in the hypotheses, we can use the `from` tactic and the elimination rule `and-elim-right`:

```
hypotheses:
  iff-sym.0 $e |- ( \iff ph0 ph1 ) $.
goal(s):
  $? |- ( \imp ph1 ph0 ) $.
  $? |- ( \imp ph0 ph1 ) $.

> from iff-sym.0, and-elim-right

hypotheses:
  iff-sym.0 $e |- ( \iff ph0 ph1 ) $.
  hyp-0 $e |- ( \imp ph1 ph0 ) $. $( new hypothesis added here $)
goal(s):
  $? |- ( \imp ph1 ph0 ) $.
  $? |- ( \imp ph0 ph1 ) $.
>
```

which applies `and-elim-right` to the hypothesis `iff-sym.0` and obtains a derived hypothesis `hyp-0`. Then the first goal can be resolved by `hyp-0`:

```
hypotheses:
  iff-sym.0 $e |- ( \iff ph0 ph1 ) $.
  hyp-0 $e |- ( \imp ph1 ph0 ) $.
goal(s):
  $? |- ( \imp ph1 ph0 ) $.
  $? |- ( \imp ph0 ph1 ) $.
```

```
> apply hyp-0

hypotheses:
  iff-sym.0 $e |- ( \iff ph0 ph1 ) $.
  hyp-0 $e |- ( \imp ph1 ph0 ) $.
goal(s):
  $? |- ( \imp ph0 ph1 ) $.
>
```

The other direction can be resolved in a similar way:

```
hypotheses:
  iff-sym.0 $e |- ( \iff ph0 ph1 ) $.
  hyp-0 $e |- ( \imp ph1 ph0 ) $.
goal(s):
  $? |- ( \imp ph0 ph1 ) $.

> from iff-sym.0, and-elim-left
...
> apply hyp-1

no goals left!
```

In the end, ITP_{ML} can generate a Metamath-verifiable proof:

```
> proof
$p |- ( \iff ph1 ph0 ) $= ph1-is-pattern ph0-is-pattern $( ... omitted $) $.
```

3.2 Proof State and Basic Tactics

The main interface of ITP_{ML} is a read-eval-print loop (REPL) that maintains a *proof state*, and accepts user-input *tactics* to transform the proof state, until all goals have been resolved.

To define the proof state, we introduce some notations first. We use $\mathcal{S}$ to denote the set of matching logic judgments, including provability judgements `|-` ... and notation judgements `#Notation` ... (as defined in Section 2.2.1). Each $s \in \mathcal{S}$ can contain *metavariables* from an infinite set $\mathcal{V}$. We use $\mathcal{T}$ to denote the set of theorems and axioms, each in the form of an inference rule. For $\tau \in \mathcal{T}$, we use $\mathit{Premises}(\tau) \in \mathcal{S}^*$ to denote the list of premises of τ and $\mathit{Conclusion}(\tau) \in \mathcal{S}$ to denote the conclusion of τ.

A *partial proof tree* π is inductively defined by the following grammar:

$$\tau \in \mathcal{T}, s \in \mathcal{S}$$
$$\pi ::= \texttt{proof}(\tau, s, \pi_1, \ldots, \pi_{|\mathit{Premises}(\tau)|}) \mid \texttt{unresolved}(s)$$

which is essentially a more compact notation for inference rules. π is called a *(complete) proof tree* if it does not contain any `unresolved` node. We denote the set of partial proof trees $\mathcal{P}$ and the set of complete proof trees $\bar{\mathcal{P}}$. We also define $\mathit{Conclusion}(\pi) \in \mathcal{S}$ to be

$$\mathit{Conclusion}(\texttt{proof}(\tau, s, \ldots)) = s \quad \mathit{Conclusion}(\texttt{unresolved}(s)) = s$$

A *proof state* is a triple $(H : \mathcal{L} \rightharpoonup \bar{\mathcal{P}}, \pi \in \mathcal{P}, \gamma \in \mathcal{S}^*)$, where

- H is a finite map from some label set $\mathcal{L}$ to complete proof trees, representing the set of (labeled) hypotheses.
- π is a partial proof tree for the initial judgement that we are trying to prove;
- γ is a list of judgements unresolved in π, which we call *goals*.

A *tactic* is simply a partial function that maps a proof state to another (or fails). To provide flexibility, ITP_{ML} supports both forward and backward proofs, and allows users to dynamically add intermediate lemmas during a proof. In this section, we discuss the related proof tactics:

- The `apply` proof tactic applies an axiom or theorem to the current proof goal and reduces it into subgoals. This accomplishes *backward proofs*.
- The `from` proof tactic applies an axiom or theorem to the hypotheses and obtain a new hypothesis. This accomplishes *forward proofs*.
- The `claim` proof tactic allows one to start a proof anywhere in between.

In the following, we discuss these tactics in more detail. For each of them, we assume the current proof state to be (H, τ, γ) with γ nonempty.

3.2.1 The `apply` tactic.

This tactic applies a theorem, axiom, or hypothesis to a goal. Suppose we have the following matching logic theorem or axiom τ, with $\mathit{Premises}(\tau) = (s_1, \ldots, s_n)$ and $\mathit{Conclusion}(\tau) = s$:

```
${ $e s_1 $. ... $e s_n $.
   τ $p s $. $}
```

`apply` τ does the following:

- Replace each metavariable in $s, s_1, \ldots, s_n$ with a fresh metavariable in $\mathcal{V}$.
- Unify s with γ_1 to get a substitution σ.
- Replace $\texttt{unresolved}(\gamma_1)$ in π by

  $$\texttt{proof}(\tau, s[\sigma], \texttt{unresolved}(s_1[\sigma]), \ldots, \texttt{unresolved}(s_n[\sigma])$$

  and get a new proof tree π'.
- Return $(H, \pi'[\sigma], (s_1[\sigma], \ldots, s_n[\sigma], \gamma_2, \ldots, \gamma_m))$, where $\pi'[\sigma]$ means applying σ to all judgements in π'.

For hypotheses, if `apply` l is used with $H(l) \in \bar{\mathcal{P}}$ defined, we first unify $\mathit{Conclusion}(H(l))$ with γ_1 to get a substitution σ. We then replace $\texttt{unresolved}(\gamma_1)$ in π with the hypothesis $H(l)[\sigma]$.

The unification algorithm we use is a modified version of the usual first-order unification algorithm by Martelli Montanari [17] that also considers the syntactical differences through the `#Notation` relation, which is detailed in Section 3.4.

3.2.2 The `from` tactic.

As a dual to `apply`, the `from` tactic applies a theorem/axiom to hypotheses instead of a goal, which allows a forward style of reasoning. The usage of `from` is

```
from l_1, ..., l_n, τ
```

where $l_1, \ldots, l_n$ are labels of hypotheses, and τ is a theorem or axiom.

The `from` tactic would then unify $\mathit{Conclusion}(H(l_i))$ with $\mathit{Premises}(\tau)_i$ for each $i \in \{1, \ldots, n\}$, assuming $|\mathit{Premises}(\tau)| = n$. Let the resulting substitution be σ. We construct a proof tree

$$\pi' := \texttt{proof}(\tau, \mathit{Conclusion}(\tau)[\sigma], H(l_1)[\sigma], \ldots, H(l_n)[\sigma])$$

and update H to $H[l \mapsto \pi']$ for some fresh $l \in \mathcal{L}$.

3.2.3 The `claim` tactic.

We often want to introduce intermediate results during a proof. This can be achieved by the `claim` tactic. For example

```
goal(s):
  ...
> claim "|- ( \imp ph0 ph0 )"
goal(s):
  [claim-0] $? |- ( \imp ph0 ph0 ) $.
  ...
```

which will add a new goal `|- ( \imp ph0 ph0 )`. The label `claim-0` then can be used as a theorem in the rest of the proof, and the new goal(s) can be proven in any order as long as there is no circular dependency between claims and the main judgement to be proved.

3.2.4 Tactic combinators.

Besides normal tactics, ITP_{ML} also supports combining tactics through tactic combinators:

- $t*$ applies t zero or more times until it fails.
- $t+$ applies t one or more times until it fails.
- $t|t'$ applies t, and if it fails, applies t'.
- $t;t'$ applies t and t' in sequence.

3.3 Metamath Proof Generation

When a complete proof tree is constructed, ITP_{ML} generates a proof object that can be directly checked by a Metamath proof checker. The trustworthiness of ITP_{ML} and its proof tactics is thus established by the generation of machine-checkable proof objects. The proof trees constructed by ITP_{ML} are already close to the formal proofs in Metamath, but there are still some gaps, which we briefly explain in this section.

Firslty, each theorem/axiom τ in Metamath has an implicit list of hypotheses on the *well-formedness* the metavariables in τ. ITP_{ML} automatically generates these well-formedness proofs according to the syntax of patterns (Definition 1).

Secondly, when applying theorems/axioms, ITP_{ML} conducts *unification modulo notations*, defined using `#Notation` (explained shortly in Section 3.4). However, Metamath proof checking conducts *syntactic unification* that requires syntactic equality. Therefore, we need to close the gap by automatically generating the *proofs for notations*, which we discuss in detail in Section 3.4.

For example, the proof tree generated Section 3.1 has the following form:

$$\texttt{proof}(\texttt{and-intro}, \texttt{|- (\and (\imp ph0 ph1) (\imp ph1 ph0))}, \ldots)$$

whereas the original proof goal is `|- ( \iff ph0 ph1 )`. When this proof is converted into a proof object in Metamath, we need to insert an additional proof for the notation $\varphi_0 \leftrightarrow \varphi_1$:

$$\frac{\texttt{|- (\and (\imp ph0 ph1) (\imp ph1 ph0))}}{\texttt{|- (\iff ph0 ph1)}}$$

With the generation of the (additional) well-formedness proofs and notation proofs, we can encode any ITP_{ML} proof tree into a Metamath proof object, by simply outputting the labels of the theorems/axioms in the tree in post-order.

3.4 Derived Notations.

Recall that in Section 2.2.1, notations are formally captured by a congruence relation `#Notation` over the syntax of patterns and other metalevel relations. In Metamath, we would need to explicitly provide a proof of notation, which can be a tedious task. ITP_{ML} reduces this work by operating on equivalence classes of patterns modulo the `#Notation` relation.

For instance, in Section 3.1, when the `apply` tactic is used, ITP_{ML} can unify `( \iff ph0 ph1 )` with `( \and ph2 ph3 )`. This is done by extending the usual first-order unification algorithm by Martelli and Montanari [17] with an extra rule that unifies terms with different function symbols if they can be rewritten to have the same function symbol through `#Notation` axioms.

Then during proof generation, we use a simple procedure to generate a Metamath proof that two patterns are syntactically equal modulo `#Notation`. We treat all notation axioms as rewrite rules directed from left to right, then we rewrite both patterns to their canonical form consisting only of the base connectives $(\bot, \rightarrow, \exists, \mu)$. If their canonical forms are equal, we generate a notation proof based on the axioms applied to rewrite these two patterns.

3.5 Automated Propositional Reasoning.

Manually proving propositional tautologies can be a considerable amount of work. For instance, the formulation of ZFC set theory in Metamath contains over a thousand propositional theorems [19].

ITP_{ML} has an automated tactic `tautology` that implements a complete procedure to generate proofs of propositional tautologies. The `tautology` tactic uses a refutation theorem proving method by using the resolution rule:

$$\frac{\varphi_1 \vee \dots \varphi_m \vee \psi \qquad \varphi'_1 \vee \dots \varphi'_n \vee \neg\psi}{\varphi_1 \vee \dots \varphi_m \vee \varphi'_1 \vee \dots \vee \varphi'_n} \text{(Resolution)}$$

where all formulas are literals (atomic propositions or negation of atomic propositions). The resolution rule is *refutation complete* [24], that is, 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 in Metamath that can be used to normalize every propositional pattern in matching logic to a CNF formula. Then the tactic will take a valid propositional pattern φ in matching logic, provably transform $\varphi \rightarrow \bot$ to CNF, and use the resolution rule to prove $(\varphi \rightarrow \bot) \rightarrow \bot$, which implies φ.

4 Related and Future Work

Research in interactive theorem proving has a long history. A class of mainstream interactive theorem provers uses the LCF approach [23] to treat types as theorems and terms with the corresponding types as proofs. This includes theorem provers based on higher-order logic [7]: Isabelle/HOL [21], HOL [27], HOL Light [12], and on various dependent type theories: Agda [3], NuPRL [1], Coq [18], Lean [9]. Many of these theorem provers are highly mature and have a large library of theorems, so ITP_{ML} is less helpful when it comes to formalizing mathematical theories in general. However, ITP_{ML} is more geared towards an orthogonal direction of generating proof objects that can be used as witnesses of correctness for the $\mathbb{K}$ framework. The proof object generation also gives ITP_{ML} an arguably smaller trust base than some of the theorem provers above such as Coq, since the proof system of matching logic is much simpler than calculus of inductive constructions.

The reference implementation of Metamath also has a built-in proof assistant [20], but it is mainly designed for encoding a proof in Metamath rather than interacting with the user to find the proof. Hence the addition of ITP_{ML} to Metamath, besides basic proving mechanisms, is a set of automated tactics specific to matching logic.

4.0.1 Future work.

In future work, we aim to continue improving the usability of ITP_{ML} by adding more proof automation. This includes a potential generalization of the notation prover to all inductively defined relations. Furthermore, a quantifier free fragment of matching logic has been shown to be decidable through a tableau method. We hope to adapt this method to produce proof objects for its correctness. In the long term, we also plan to integrate ITP_{ML} with other provers of $\mathbb{K}$ to produce a uniform platform to reason not only in matching logic, but in $\mathbb{K}$ directly.

References

[1] Stuart F Allen, Mark Bickford, Robert L Constable, Richard Eaton, Christoph Kreitz, Lori Lorigo, and Evan Moran. Innovations in computational type theory using Nuprl. *Journal of Applied Logic*, 4(4):428–469, 2006.

[2] Denis Bogdanas and Grigore Roşu. K-Java: A complete semantics of Java. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'15)*, pages 445–456, 2015.

[3] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda – a functional language with dependent types. In *International Conference on Theorem Proving in Higher Order Logics*, pages 73–78. Springer, 2009.

[4] Xiaohong Chen, Zhengyao Lin, Minh-Thai Trinh, and Grigore Rosu. Towards a trustworthy semantics-based language framework via proof generation. In *Proceedings of the 33th International Conference on Computer Aided Verification (CAV'21)*, 2021. `http://hdl.handle.net/2142/107794`.

[5] Xiaohong Chen, Dorel Lucanu, and Grigore Rosu. Matching logic explained. Technical report, University of Illinois at Urbana-Champaign, 2021. `http://hdl.handle.net/2142/107794`.

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

[7] Alonzo Church. A formulation of the simple theory of types. *The journal of symbolic logic*, 5(2):56–68, 1940.

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

[9] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In Amy P. Felty and Aart Middeldorp, editors, *Proceedings of the 25th International Conference on Automated Deduction Automated Deduction (CADE'15)*, pages 378–388, Cham, 2015. Springer International Publishing.

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

[11] Dwight Guth. A formal semantics of Python 3.3. 2013.

[12] John Harrison. HOL light: An overview. In *International Conference on Theorem Proving in Higher Order Logics*, pages 60–66. Springer, 2009.

[13] 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.

[14] K Team. Matching logic proof checker. GitHub page https://github.com/kframework/matching-logic-proof-checker/blob/main/theory/matching-logic-250-loc.mm, 2021.

[15] Giacomo Lenzi. The modal μ-calculus: A survey. *Task quarterly*, 9(3):293–316, 2005.

[16] Zhengyao Lin, Xiaohong Chen, Minh-Thai Trinh, John Wang, and Grigore Rosu. Trustworthy program verification via proof generation. Technical report, University of Illinois at Urbana-Champaign, 2021. http://hdl.handle.net/2142/110344.

[17] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(2):258–282, 1982.

[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*. 2019.

[21] Tobias Nipkow, Markus Wenzel, and Lawrence Paulson. *Isabelle/HOL: A proof assistant for higher-order logic*. Springer, 2002.

[22] 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.

[23] Lawrence C Paulson. *Logic and computation: interactive proof with Cambridge LCF*, volume 2. Cambridge University Press, 1990.

[24] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1):23–41, 1965.

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

[26] 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.

[27] Konrad Slind and Michael Norrish. A brief overview of HOL4. In *International Conference on Theorem Proving in Higher Order Logics*, pages 28–32. Springer, 2008.

[28] The Isabelle development team. Isabelle, 2018. https://isabelle.in.tum.de/.