A Practical Trustworthy Language Framework

Thesis Proposal

Xiaohong Chen
University of Illinois at Urbana-Champaign
xc3@illinois.edu

Contents

1 Introduction 1

2 Overview of Proposed Method 3

3 Current Results 5
3.1 Proposal of Matching Logic 5
3.2 Expressiveness of Matching Logic 7
3.3 Automated Fixpoint Reasoning for Matching Logic 10
3.4 Known Results about Matching Logic Complete Deduction 11

4 Proposed Work 12
4.1 Matching Logic Proof Checker 12
4.1.1 Metamath Overview 13
4.1.2 Matching Logic Proof Checker Implemented in Metamath 14
4.1.3 Generating Proof Objects for Program Execution and Verification 15
4.2 Matching Logic Complete Deduction 16
4.3 Matching Logic Decidable Fragments 16
4.4 Defining Logical Systems in Matching Logic 19

5 Conclusion 20

1 Introduction

A *language framework* is an artifact that allows language designers to define the formal syntax and semantics of their programming languages using an intuitive and easy-to-understand meta-language. Once the formal syntax and semantics definition is given, the framework automatically generates all language tools of that language from its formal definition in a correct-by-construction manner, at no additional costs. By language tools, I mean both an *implementation* of the language that often consists of a parser and an interpreter and/or a compiler so as to execute programs written in that language and *formal analysis tools* such as a deductive program verifier and a bounded model checker. The above is called the *vision of an ideal language framework*, illustrated in Figure 1.

The $\mathbb{K}$ language framework (`http://kframework.org`) is a pursuer of the above ideal vision and has achieved much success in practice. Using $\mathbb{K}$, the formal definitions of many real-world languages

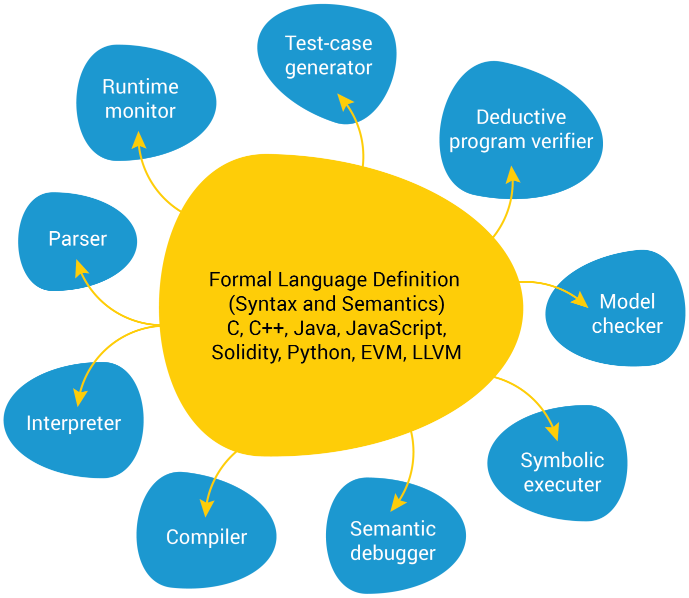
Test-case generator
Runtime monitor
Deductive program verifier
Parser
Formal Language Definition (Syntax and Semantics) C, C++, Java, JavaScript, Solidity, Python, EVM, LLVM
Model checker
Interpreter
Symbolic executer
Compiler
Semantic debugger

Figure 1: Vision of an ideal language framework.

have been defined and their implementations and formal analysis tools have been automatically generated. As of this writing, $\mathbb{K}$ has successfully been used to define the following programming languages: C [20], Java [3], JavaScript [28], Python [19], Ethereum virtual machine bytecode [21], x86-64 [13] and to generate their language tools. There are commercial products based on the tools generated by $\mathbb{K}$ (`https://runtimeverification.com/`).

$\mathbb{K}$ has had several different implementations, all of which are complex. The most recent $\mathbb{K}$ implementation (`https://github.com/kframework/k`) is developed in several languages including Haskell, C^{++}, Scala, and OCaml, with more than 500,000 lines of code. It consists of a front end that compiles a language definition written in $\mathbb{K}$ using an intuitive and compact human-readable meta-language into one that is written in a lower level language (called Kore) where syntactic sugar, user-defined notations, comments, shortcuts, and other front end syntax are eliminated. The resulting lower-level Kore definition is not as human-readable as the original $\mathbb{K}$ definition but it can be easily understood and processed by $\mathbb{K}$'s back ends. These back ends implement complex algorithms to generate the language tools and to conduct various tasks such as program execution, formal verification, and symbolic execution.

A serious question is how to guarantee the *correctness* of a language framework implementation, such as $\mathbb{K}$, which has a large code base and implements complicated algorithms. In other words, if we define a programming language in $\mathbb{K}$ and use $\mathbb{K}$ to execute a program written in that language, to what extend can we trust the execution result that $\mathbb{K}$ returns? Similar questions can be asked not only for program execution but also for formal verification, symbolic execution, bounded model checking, and many other formal analysis tools that $\mathbb{K}$ generates for the language. In short, what is the trust base of $\mathbb{K}$?

Currently, to trust the execution result that $\mathbb{K}$ returns (or any other execution and/or analysis tasks that $\mathbb{K}$ conducts) we need to trust the entire code base of $\mathbb{K}$, which is large and certainly contains bugs. My research goal is to make that trust base smaller, from more than 500,000 lines of code written in four different programming languages, to a small *proof checker* [23] that has only 245 lines of code, *without* the need to re-implement the existing $\mathbb{K}$ front and back ends. This way, I make $\mathbb{K}$ a *trustworthy* and *practical* language framework.

In this proposal, I discuss the methodology to accomplish the above goal. I first give an overview of the proposed method in Section 2. Then I discuss the work that has been done in Section 3 and the proposed future work in Section 4. A timeline to finish the proposed work in 12 months is given at the end of Section 2.

2 Overview of Proposed Method

The proposed method is based on a *logical foundation* of $\mathbb{K}$.

In short, I design a mathematical logic called *matching logic* that aims at serving as the logical foundation of $\mathbb{K}$. By that, I mean that each $\mathbb{K}$ definition of a programming language L yields a *matching logic theory* Γ^L that includes the definition of the syntax and semantics of L. Further, each task that $\mathbb{K}$ does for the language L is expressed by a matching logic formula φ, called a *pattern*. For example, the task where we use $\mathbb{K}$ to run the following C program snippet

$$\mathsf{SUM} \equiv \mathsf{while(!n)\ \{\ s{=}s{+}n;\ n{=}n{-}1;\ \}}$$

on a state where n is 100 and s is 0, and that $\mathbb{K}$ returns the value 5050, can be expressed by the following matching logic pattern, intuitively:

$$\Gamma^{\mathrm{C}} \vdash \left\langle \langle \mathsf{SUM} \rangle_{\mathsf{code}} \, \langle \mathsf{n} \mapsto 100, \mathsf{s} \mapsto 0 \rangle_{\mathsf{state}} \right\rangle_{\mathsf{cfg}} \Rightarrow^{*} \left\langle \langle \cdot \rangle_{\mathsf{code}} \, \langle \mathsf{n} \mapsto 0, \mathsf{s} \mapsto 5050 \rangle_{\mathsf{state}} \right\rangle_{\mathsf{cfg}} \qquad (1)$$

where Γ^{C} is a matching logic theory that includes the formal syntax and semantics of C and $\left\langle \langle c \rangle_{\mathsf{code}} \, \langle \rho \rangle_{\mathsf{state}} \right\rangle_{\mathsf{cfg}}$ is the (matching logic) pattern that represents a computation configuration[1] where the current code is c and the program state is ρ. The symbol "$\Rightarrow^{*}$" specifies the rewriting/reachability relation between the two configurations, meaning that the first configuration eventually reaches the second configuration within finitely many execution steps (Section 3.2). Its semantics is defined by the matching logic theory Γ^{C} using patterns/axioms.

The key idea to achieve a small trust base for $\mathbb{K}$ is as follows. We encode the correctness of $\mathbb{K}$ conducting a task, say executing the above C program SUM and returning the value 5050 using C semantics, into a *matching logic formal proof* $\Gamma^{\mathrm{C}} \vdash \varphi$ like Equation (1), which derives φ from Γ^L using one fixed *matching logic proof system* (Figure 2). The formal proof $\Gamma^{\mathrm{C}} \vdash \varphi$ is further encoded into a *proof object* that is annotated with the detailed proof steps, so the proof object witnesses the formal proof and can be quickly checked by a *matching logic proof checker*, which implements the proof rules in the proof system in Figure 2. This way, the correctness of $\mathbb{K}$ conducting one task is reduced to the correctness of the proof checker checking one proof object. This is preferable, because the proof checker is much smaller than the entire $\mathbb{K}$ implementations. I implemented a prototype proof checker in [23] using the Metamath language [25] in 245 lines of code.

[1] The actual computation configurations for C are more complicated, with more than a hundred *configuration cells* that hold the necessary semantic information needed for program execution. Here, we simplify it into a configuration with only two cells (code that holds the computation and state that holds the program state) for the sake of presentation.

The key observation is that correctness is established for each individual task that $\mathbb{K}$ does on a *case-by-case* basis, and not for the entire $\mathbb{K}$ code base. If the bugs in $\mathbb{K}$ are not triggered when $\mathbb{K}$ conducts the task, the correctness is not compromised and a proof object can be generated. If the bugs are triggered, the correctness is lost and there does not exist a corresponding proof object for the wrong result that $\mathbb{K}$ returns. Therefore, proof objects serve as *correctness certificates* that need to be issued for each tasks that $\mathbb{K}$ carries out.

The above proposed approach can be broke down into the following deliverables:

1. a proposal of the logical foundation—matching logic—of $\mathbb{K}$; this should include the syntax, the semantics, and the proof system of matching logic;

2. a collection of matching logic theories that define the common logical systems that are used to specify the formal semantics of programming languages and program properties; this guarantees that matching logic has the expressiveness that is needed to encode tasks that a language framework should usually support;

3. a completeness result of matching logic proof system; this establishes the connection between matching logic semantics and its formal deduction so that we know the proof rules proposed in item 1 are not ad-hoc;

4. decidable fragment of matching logic and their corresponding decision procedures; this brings up the level of automation in the current $\mathbb{K}$ implementations;

5. a matching logic proof checker that is small and fast and a proof object generator that encodes the tasks that $\mathbb{K}$ conducts such as program execution and formal verification into proof objects that can be proof-checked by the proof checker.

Publications and timeline. The deliverables that have been finished are discussed in Section 3 and those that are planned for the future are discussed in Section 4. I summarize them in the following. For each item, I associate the corresponding publications (in **black**) as well as the planned submissions (in **green**).

- Current results that have been published include:
  1. **[6, LICS2019]**: the proposal of matching logic and a 13-rule Hilbert-style proof system (Section 3.1 and Figure 2);
  2. results showing that many important logical systems can be defined as matching logic theories (Section 3.2 and Figure 3), including
     (a) **[6, LICS2019]**: FOL, FOL with least fixpoints, modal logic, temporal logics, dynamic logic, modal μ-calculus, and reachability logic;
     (b) **[7, ICFP2020]**: λ-calculus, π-calculus, and type systems;
     (c) **[5, TechRep2020]**: initial algebra semantics, planned for **[LICS2021]**;
  3. **[8, OOPSLA2020]**: a prototype automated theorem prover of matching logic, focusing on fixpoitn reasoning (Section 3.3);
  4. **[6, LICS2019]**: some preliminary completeness results (Section 3.4).

- Future work (including work that has partial results) includes:

1. a matching logic proof checker and a procedure to generate proof objects (Section 4.1), where
   (a) an prototype proof checker implemented in Metamath and the proof object generation for a simple imperative language are planned for **[CAV2021]**;
   (b) a full integration of proof object generation for program execution and formal verification into $\mathbb{K}$ is planned for **[PLDI2022]**;
2. a completeness theorem based on Henkin semantics (Section 4.2), planned for **[LICS2021]**;
3. decidable fragments of matching logic and their decision procedures (Section 4.3), planned for **[LICS2021]**;
4. matching logic theories that define separation logic and hyper linear temporal logic (Section 4.4), planned for **[POPL2022]** and **[LICS2021]**, respectively.

3 Current Results

In this section, I summarize the main research results obtained so far.

3.1 Proposal of Matching Logic

I proposed with my supervisor professor Grigore Roşu *matching logic* in its full generality [6]. Matching logic serves as the unifying logical foundation of the proposed trustworthy language framework. Its syntax is parametric in a *signature* that consists of user-provided *symbols* and defines formulas, called *patterns*, to uniformly represent data structures, mathematical objects, computations, program configurations, transition relations, dynamic properties of programs, and so on. Its semantics defines *models*, which can be constrained by a set of user-provided *pattern axioms.* A matching logic *proof system* is used to derive new patterns from a given set of axioms.

The power of matching logic lies in its simplicity and expressiveness. In terms of simplicity, matching logic follows a minimalism design. For example, the syntax of matching logic patterns, shown below, has only 8 syntactic constructs that define the most basic concepts that are necessary to serve as the logical foundation of a language framework:

$$
\begin{aligned}
\varphi ::= {} & x && \text{// element variables} \\
\mid {} & X && \text{// set variables} \\
\mid {} & \sigma && \text{// symbols (from a user-provided signature)} \\
\mid {} & \varphi_1\,\varphi_2 && \text{// application} \\
\mid {} & \bot && \text{// bottom} \\
\mid {} & \varphi_1 \to \varphi_2 && \text{// implication} \\
\mid {} & \exists x \,.\, \varphi && \text{// quantification} \\
\mid {} & \mu X \,.\, \varphi && \text{// least fixpoints}
\end{aligned}
$$

Every syntactic construct in the above has a unique purpose, which I explain intuitively. Element variables are used to refer to the individual elements in the models, like FOL variables. Set variables are used to refer to the sets of elements in the models, like propositional variables in modal logic. Symbols are provided by the users and are used to represent constructors, functions, predicates, relations, and so on, whose actual semantics is determined by the axioms and theories. An application

FOL Reasoning	(Tautology)	$\varphi \quad$ if φ is a tautology
	(Modus Ponens)	$\dfrac{\varphi_1 \quad \varphi_1 \to \varphi_2}{\varphi_2}$
	($\exists$-Quantifier)	$\varphi[y/x] \to \exists x \,.\, \varphi$
	($\exists$-Generalization)	$\dfrac{\varphi_1 \to \varphi_2}{(\exists x . \varphi_1) \to \varphi_2}$ if $x \notin FV(\varphi_2)$
Frame Reasoning	(Propagation$_\bot$)	$C[\bot] \to \bot$
	(Propagation$_\vee$)	$C[\varphi_1 \vee \varphi_2] \to C[\varphi_1] \vee C[\varphi_2]$
	(Propagation$_\exists$)	$C[\exists x \,.\, \varphi] \to \exists x \,.\, C[\varphi] \quad$ if $x \notin FV(C)$
	(Framing)	$\dfrac{\varphi_1 \to \varphi_2}{C[\varphi_1] \to C[\varphi_2]}$
Fixpoint Reasoning	(Substitution)	$\dfrac{\varphi}{\varphi[\psi/X]}$
	(PreFixpoint)	$\varphi[(\mu X \,.\, \varphi)/X] \to \mu X \,.\, \varphi$
	(Knaster-Tarski)	$\dfrac{\varphi[\psi/X] \to \psi}{\mu X \,.\, \varphi \to \psi}$
Technical Rules	(Existence)	$\exists x \,.\, x$
	(Singleton)	$\neg\,(C_1[x \wedge \varphi] \wedge C_2[x \wedge \neg\varphi])$

where $C[\varphi]$ denotes application $\varphi\,\psi$ or $\psi\,\varphi$ for any ψ.

Figure 2: Matching logic has a Hilbert-style proof system with 13 proof rules.

construct is used to apply a symbol (or any pattern in general) to an argument pattern. Bottom and implication allow us to build logical constraints. Quantification allows us to create abstraction. Least fixpoints allow us to define induction and recursion.

Besides the syntax, matching logic has a 13-rule Hilbert-style proof system (Figure 2) that derives proof judgments of the form $\Gamma \vdash \varphi$, meaning that φ can be proved using the proof system from the axioms in Γ.

The simplicity of matching logic means that the more involved concepts such as sorts, many-sorted functions, many-sorted predicates, order-sorted structures, subsort overloading, types, dependent types, parametric types, and so on are not built into it. Instead, they can be defined in an axiomatic way by matching logic theories using symbols and axioms. I explain it in Section 3.2.

I emphasize the simplicity of matching logic because it makes *proof checking* simple. A proof checker is a decision procedure whose input is a *proof object* that encodes a formal proof $\Gamma \vdash \varphi$. The proof checker checks whether the given proof object is valid. Thanks to the simplicity of matching logic syntax and its proof system, a matching logic proof checker can be implemented in 245 LOC in Metamath [23]. In Section 4.1, I explain how the proof checker serves as a small trust base of the proposed trustworthy language framework that is accessible to the users.

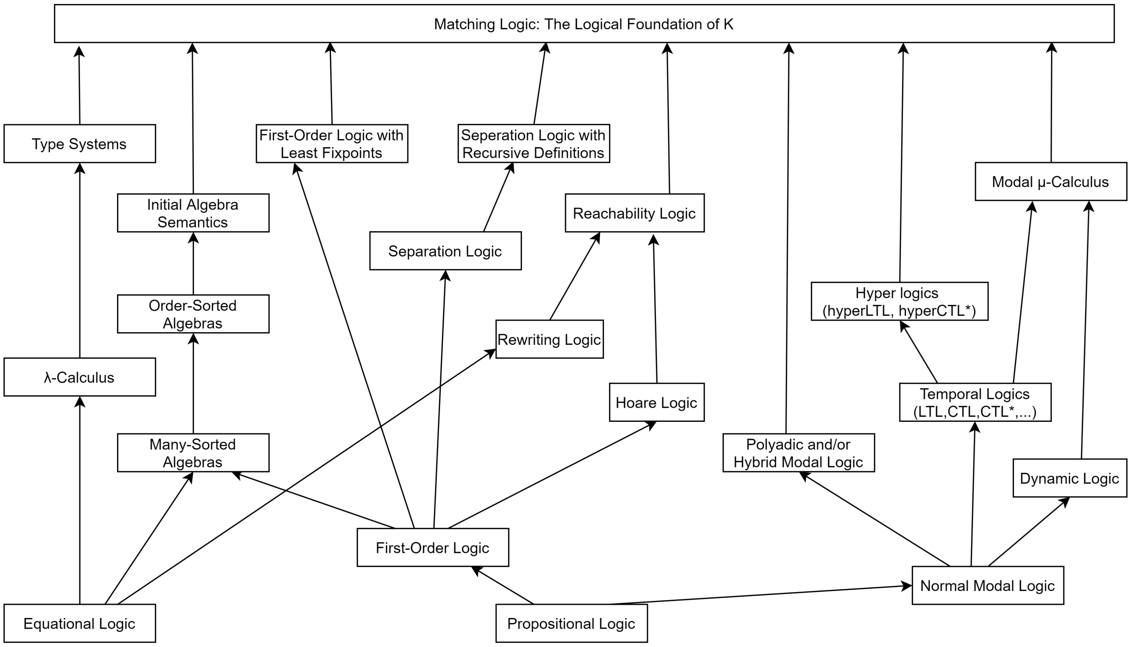
Matching Logic: The Logical Foundation of K
Type Systems
First-Order Logic with Least Fixpoints
Seperation Logic with Recursive Definitions
Modal μ-Calculus
Initial Algebra Semantics
Reachability Logic
Separation Logic
Hyper logics (hyperLTL, hyperCTL*)
Order-Sorted Algebras
Rewriting Logic
λ-Calculus
Hoare Logic
Temporal Logics (LTL,CTL,CTL*,...)
Many-Sorted Algebras
Polyadic and/or Hybrid Modal Logic
Dynamic Logic
First-Order Logic
Normal Modal Logic
Equational Logic
Propositional Logic

Figure 3: Many logical systems can be captured by matching logic theories (the figure includes both published and future work).

3.2 Expressiveness of Matching Logic

Matching logic is expressive. Many important logical systems and/or formalisms can be defined as matching logic theories. In Figure 3, I summary some common logical systems that have been defined in matching logic. With the powerful expressiveness, matching logic can serve a powerful specification language that allows us to specify both systems and their properties.

In the following, I discuss four logical systems and/or formalisms and their matching logic theories: FOL with least fixpoints (LFP), reachability logic, type systems, and initial algebra semantics. They are representative approaches to designing formal language semantics. In the LFP section, I will show how to define recursive relations and/or predicates using the least fixpoint operator μ of matching logic. In the reachability logic section, I will explain how formal verification, including the classical Hoare-style verification using Hoare triples, can be expressed using reachability rules, which can then be defined using matching logic patterns, allowing matching logic to deal with formal verification of all programming languages based on their formal semantics. In the type systems section, I will explain how binders such as λ in λ-calculus can be defined using matching logic, where the binding behavior of λ and any other binders is obtained for free from the binding behavior of the built-in binder $\exists$ in matching logic. Finally, in the initial algebra semantics, I will firstly show how sorts, many-sorted algebras, and order-sorted algebras can be axiomatically defined in matching logic and then discuss the matching logic definition of initial algebras.

FOL with least fixpoints (LFP). LFP is a popular system that is used to specify and reason about induction, recursion, and fixpoints. LFP extends the classical first-order logic by *fixpoints* and allows formulas of the form $[\mathbf{lfp}_{R,x}\varphi](t)$, where R is a *recursive relation symbol*, x is an argument of R, φ is a formula where R and x can (recursively) occur, and t is a term. Here, $[\mathbf{lfp}_{R,x}\varphi]$ denotes the smallest relation R such that $\forall x \,.\, R(x) = \varphi$.

LFP defines fixpoints differently than matching logic, which only allows the definitions of *recursive sets* using the least fixpoint operator μ: intuitively, the matching logic pattern $\mu X \,.\, \varphi$ denotes the smallest set X such that $X = \varphi$. To capture LFP recursive relations, we need to reduce them to recursive sets.

In [6], I showed that $[\mathbf{lfp}_{R,x}\varphi]$ can be defined in terms of its *indicator set* I_R, which is the set of arguments y such that $[\mathbf{lfp}_{R,x}](y)$ holds. The indicator set I_R is a recursive set and can be defined by the following pattern:

$$I_R = \mu R \,.\, \exists x \,.\, x \wedge \varphi$$

where, intuitively, $\exists x \,.\, x \wedge \varphi$ denotes the set of all x such that φ holds. Here, we tacitly regard all occurrences of $R(t)$ in φ as the predicates $t \in R$, which state that t belongs to the indicator set R. This way, I proved that LFP can be defined in matching logic and I_R defined above yields the same semantics as $[\mathbf{lfp}_{R,x}]$ in LFP.

Reachability logic and Hoare logic. Reachability logic aims at sound and (relatively) complete formal verification of all programming languages based on their operational semantics. In reachability logic, the formal semantics of a language L is defined by a set of *rewrite rules* $S = \{\varphi_i \Rightarrow \varphi_i' \mid 1 \leq i \leq n\}$, where φ_i and φ_i' are patterns matched by program states, called *configurations*. One fixed set of *reachability proof rules* is used to derive reachability judgments of the form $S \vdash \psi \Rightarrow \psi'$, meaning that any configurations that matches ψ will eventually reach ψ', unless the execution diverges; this yields *partial correctness*. As a special instance, a Hoare triple $\{\varphi_{pre}\}\ \mathit{code}\ \{\varphi_{post}\}$ can be expressed by the following reachability rule:

$$\langle \langle \mathit{code} \rangle_{\mathsf{code}} \cdots \rangle_{\mathsf{cfg}} \wedge \varphi_{pre} \Rightarrow \langle \langle \mathit{skip} \rangle_{\mathsf{code}} \cdots \rangle_{\mathsf{cfg}} \wedge \varphi_{post}$$

where the left-hand side consists of the program "*code*" and the pre-condition φ_{pre} and the right-hand side consists of no remaining code (represented by "*skip*") and the post-condition φ_{post}. An important characteristic of reachability logic is that it uses one *fixed* set of proof rules to obtain sound and (relative) complete formal verification for all programming languages, given that their formal semantics are defined by a set of rewrite rules. Reachability has been the logical foundation of the current $\mathbb{K}$'s formal verification tools.

In [6], I proved that reachability logic can be defined in matching logic and all reachability proof rules can be proved as matching logic theorems using the proof system in Figure 2. In particular, reachability rules are defined as follows:

$$\varphi_1 \Rightarrow \varphi_2 \equiv \varphi_1 \rightarrow (\nu X \,.\, \varphi_2 \vee \bullet X)$$

where ν is the greatest fixpoint operator that can be defined from the least fixpoint operator μ and $\bullet$ is a matching logic symbol called "one-path next": $\bullet X$ denotes the set of states that have (at least) a next state that matches X. I showed that the above definition works and yields the same partial-correctness semantics of reachability rules. This way, I proved that matching logic can replace reachability logic as the more uniform logical foundation of $\mathbb{K}$ and its formal verification tools.

Type systems. λ-calculus [9] and type systems are one of the foundations of computation. In λ-calculus, computations are defined in terms of *function abstraction* $\lambda x\,.\,e$, where λ is a binder that binds x in the function body e. Function application is defined by the following famous (β) rule:

$$(\beta) \quad (\lambda x\,.\,e)e' = e[e'/x]$$

In [7], I proved that λ-calculus and many type systems can be defined in matching logic. The main difficulty lies in the treatment of *binders*. Indeed, if we simply define λ as a binary function, then the binding behavior of λ such as α-equivalence and capture-avoiding substitution needs to be defined separately. If there are multiple binders, each of them needs to have its own α-equivalence and capture-avoiding substitution defined separately, leading to duplication. I proposed a novel definition of binders where the binding behavior is obtained for free by the built-in binder $\exists$ in matching logic:

$$\lambda x\,.\,e \equiv \mathsf{lambda}\ \underbrace{(\mathsf{intension}\ (\exists x\,.\ \langle x, e\rangle))}_{\text{graph of function } x \mapsto e}$$

Intuitively, $\mathsf{intension}\ (\exists x\,.\ \langle x, e\rangle)$ returns the graph of the function that maps x to e. Note that x is bound by the $\exists$ binder in the above, so the binding behavior of λ is inherited from the binding behavior of $\exists$. Then, I applied the same idea to other logical systems with binders, including type systems such as the pure type systems [2,24,32] and System F [4,15] as well as other systems with binders such as π-calculus [26].

Initial algebra semantics. Initial algebra semantics [17] is a main approach to formal language semantics based on algebraic specifications and their initial models. It has led to extensive research on its theories and applications [1,12,14,18,29,33]. The key idea is to define the sorts of data and the operations on the data as an algebraic specification E. Among all algebras that satisfy E, there is an *initial algebra* I, unique up to isomorphism, such that for any other algebra A satisfying E there is a unique morphism $h_A\colon I \to A$. In this view, the syntax of a programming language forms an initial algebra and its formal semantics is a way to associate the syntax with the *intended* semantic model (algebra). The unique morphism h_A *is* the semantic function mapping syntax to semantics. Initiality has a close relationship with *induction*. Since program syntax is often defined inductively, the initial algebra I enjoys the *principle of induction*, which can then be mapped to the semantic models through the unique morphisms.

In [5], I proved that initial algebra semantics can be defined in matching logic. Firstly, I showed that many-sorted and order-sorted algebras can be defined in matching logic in an axiomatic way. Formally speaking, a many-sorted algebra A consists of a set of sorts s_1, s_2, … whose corresponding carrier sets in A are written A_{s_1}, A_{s_2}, …, respectively. In addition, A includes a set of sorted functions $f\colon s_1 \times \ldots s_n \to s$ whose interpretations are functions $f_A\colon A_{s_1} \times \cdots \times A_{s_n} \to A_s$. Matching logic is an unsorted logic and has no built-in support for sorts, but it can define sorts and the sorts related concepts in an axiomatically way. For each sort s, we define a matching logic symbol also written s to represent its sort name. Then, we introduce an *inhabitant symbol* $[\![\_]\!]$ to get the carrier set of a sort s by applying it to s as in $[\![\_]\!]\, s$, or written $[\![s]\!]$. The nonempty-ness of the carrier sets and the signatures of many-sorted functions can be axiomatized by patterns as follows:

$$[\![s]\!] \neq \bot \qquad // \text{ the carrier set of } s \text{ is nonempty}$$

$$\forall x_1{:}s_1 \ldots \forall x_n{:}s_n\,.\,\exists y{:}s\,.\,f\,x_1\,\ldots\,x_n = y \qquad // \ f \text{ is a many-sorted function}$$

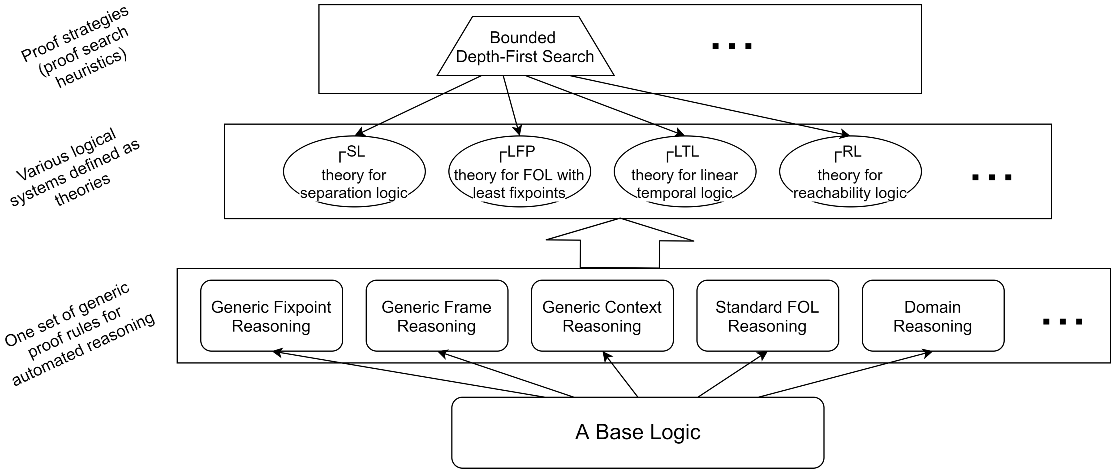
Proof strategies (proof search heuristics)
Bounded Depth-First Search
...
Various logical systems defined as theories
Γ^SL theory for separation logic
Γ^LFP theory for FOL with least fixpoints
Γ^LTL theory for linear temporal logic
Γ^RL theory for reachability logic
...
One set of generic proof rules for automated reasoning
Generic Fixpoint Reasoning
Generic Frame Reasoning
Generic Context Reasoning
Standard FOL Reasoning
Domain Reasoning
...
A Base Logic

Figure 4: A unified automated proof framework based on matching logic.

where the *sorted quantification* can also be defined by patterns:

$$\forall x{:}s \,.\, \varphi \equiv \forall x \,.\, x \in [\![s]\!] \to \varphi \qquad\qquad \exists x{:}s \,.\, \varphi \equiv \exists x \,.\, x \in [\![s]\!] \land \varphi$$

Order-sorted algebras can be defined in a similar, axiomatic way. In particular, the subsorting relation $s_1 < s_2$ is axiomatized by the following pattern:

$$[\![s_1]\!] \subseteq [\![s_2]\!] \qquad\qquad // \; s_1 \text{ is a subsort of } s_2$$

Initiality is defined using the least fixpoint operator μ. For example, the initial algebra that has one sort *Nat* and two constructors *zero* and *succ* can be defined by the following patterns following the famous "no-junk no-confusion" slogan:

$$\begin{aligned}
&[\![Nat]\!] = \mu N \,.\, zero \lor succ(N) && // \text{ no-junk}\\
&\forall x{:}Nat \,.\, zero \neq succ(x) && // \text{ no-confusion, different constructors}\\
&\forall x{:}Nat \,.\, \forall y{:}Nat \,.\, succ(x) = succ(y) \to x = y && // \text{ no-confusion, same constructor}
\end{aligned}$$

This way, I internalized initial algebra semantics in matching logic.

3.3 Automated Fixpoint Reasoning for Matching Logic

Given the expressiveness of matching logic, it is highly desirable that we develop an *automated theorem prover* for matching logic, as it can be instantiated by the various logical theories that we defined in Section 3.2 and obtained specialized provers for many logical systems.

In [8], I developed a prototype matching logic prover, with a focus on the reasoning about fixpoints and contexts. Figure 4 illustrates the prover architecture. It is based on matching logic, upon which a set of generic reasoning modules are implemented for fixpoint reasoning, frame reasoning, context reasoning, FOL reasoning, and domain reasoning using SMT solvers. Automated reasoning is implemented as a *proof search* over the proof rules of these reasoning modules.

The prover can be instantiated by a matching logic theory, resulting in a specialized prover for that theory. For example, we can instantiate the prover by a theory Γ^{SL} that defines separation logic [30] and the result is a separation logic prover. In [8], I carried out an experiment and instantiated the prover by four matching logic theories that define LFP, separation logic, linear temporal logic (LTL), and reachability logic, respectively. The experiment results were promising, showing that the specialized provers have competitive effectiveness compared to the state of the art. For example, the separation logic prover instantiated by Γ^{SL} was able to resolved 265 out of 280 obligations in the official SL-COMP 2019 benchmark, which would have ranked it the third place if it participated in the competition.

3.4 Known Results about Matching Logic Complete Deduction

A completeness theorem establishes the connection between the semantic validity relation $\Gamma \vDash \varphi$ and the provability relation $\Gamma \vdash \varphi$. Intuitively, $\Gamma \vDash \varphi$ states that all matching logic models that satisfy the axioms in Γ also satisfy φ, while $\Gamma \vdash \varphi$ states that there exists a formal proof using the proof system in Figure 2 to prove φ from the axioms in Γ. The connection between them can be illustrated as the following soundness/completeness properties:

$$\Gamma \vDash \varphi \xleftrightarrow[\text{completeness}]{\text{soundness}} \Gamma \vdash \varphi$$

Soundness has been proved in [6] but completeness is more involved.

In general, matching logic does not have complete deduction. This means that for any (effective) proof system, the above completeness property will fail for some Γ and φ. The reason for that is that matching logic supports FOL-style quantification as well as the least fixpoints. We can thus define a theory Γ^{Nat} that captures precisely the standard model of natural numbers with addition and multiplication. By Godel's first incompleteness theorem [16], there does not exists an effective proof system that is complete for the theory Γ^{Nat}. However, it is still possible that the proof system in Figure 2 is complete for a fragment of matching logic.

In Table 1, I summarize the known results and open problems about matching logic complete deduction (as well as its decidable fragments and small model properties, which will be discussed as proposed work in Section 4). Since matching logic does not have complete deduction in general, I studied its *syntactic fragments* that include only a subset of the syntactic constructs of matching logic. In Table 1, I use $\{\}$ to denote the fragment of matching logic that includes only symbols, application, and the propositional connectives $\bot$ and $\to$. I use exists to mean element variables and the $\exists$ quantification and use mu to mean set variables and the μ least fixpoint operator. Therefore, $\{\mathsf{exists}, \mathsf{mu}\}$ denotes the full matching logic with all the syntactic constructs.

I call the fragment denoted by $\{\mathsf{exists}\}$ the *fixpoint-free fragment*, because it excludes set variables and fixpoints. In [6], I proved two completeness results for the fixpoint-free fragment of matching logic. The first completeness result shows that all valid patterns can be proved from the proof system in Figure 2.

Theorem 1 *Consider the fixpoint-free fragment of matching logic. The proof system in Figure 2 is complete for the empty theory, that is,* $\emptyset \vDash \varphi$ *implies* $\emptyset \vdash \varphi$*.*

Then, I pushed the above theorem further to theories where *equality* is (axiomatically) defined.

Theorem 2 *Under the condition in Theorem 1, the proof system is complete for any theory* Γ *that defines equality.*

Table 1: A summary of the known results (denoted ✓and ✗) and open problems (denoted ?) about matching logic, including its complete deduction, decidable fragments, and small model properties.

Properties	**Syntactic Fragments of Matching Logic**			
	{}	{mu}	{exists}	{mu, exists}
Completeness (for the empty theory)	✓	?	✓	?
Completeness (for all theories)	?	?	✓[1]	✗
Decidability (for the empty theory)	$\boxed{\checkmark}$	$\boxed{\checkmark}$	?	?
Decidability (for all theories)	?	?	✗	✗
Small model property (for the empty theory)	✓	$\boxed{\checkmark}$	?	?
Small model property (for all theories)	✗	✗	✗	✗

The framed check marks $\boxed{\checkmark}$ indicate results that have been proved but not yet published, which will be discussed as proposed work in Section 4.

ˆ1. Only proved for theories that define equality, inspired by [31].

These completeness results are not of theoretical interest only. They are evidence that show that the proof system in Figure 2 is not ad-hoc. Instead, it is closely related to the semantics and the validity relation.

Future work on the completeness of matching logic based on Henkin semantics is explained in Section 4.2. Future work on the decidable fragments of matching logic as well as the small model properties is explained in Section 4.3.

4 Proposed Work

In this section, I discuss technical details of the proposed research.

4.1 Matching Logic Proof Checker

As shown in Figure 2, the matching logic proof system is a Hilbert-style proof system that derives judgments of the form $\Gamma \vdash \varphi$. Formally, a Hilbert-style proof within a theory Γ has the form $\langle \Gamma : \varphi_1 \varphi_2 \cdots \varphi_k \rangle$ such that for each $1 \leq i \leq k$, φ_i is an axiom (in the proof system or in Γ) or it is the result of applying a proof rule on the patterns that appear before φ_i. We can annotate how each φ_i is proved along with the Hilbert-style proof and the resulting annotated proof, written $\langle \Gamma : \varphi_1, \ldots, \varphi_k \rangle_{\mathsf{obj}}$, is called a *proof object*. A *proof checker* is an algorithm that takes a proof object as input and checks whether it encodes a correct Hilbert-style proof. The following are the key properties that a proof checker and the proof objects should satisfy:

Correctness. If $\mathsf{ProofCheck}\left(\langle \Gamma : \varphi_1, \ldots, \varphi_k \rangle_{\mathsf{obj}}\right) = \text{true}$ then $\Gamma \vdash \varphi_k$.

Effectiveness. If $\Gamma \vdash \varphi_k$ then there exist a proof object $\langle \Gamma : \varphi_1, \ldots, \varphi_k \rangle_{\mathsf{obj}}$ such that

$$\mathsf{ProofCheck}\left(\langle \Gamma : \varphi_1, \ldots, \varphi_k \rangle_{\mathsf{obj}}\right) = \text{true}$$

In the following, I briefly discuss how to create proof objects $\langle \Gamma : \varphi_1, \dots, \varphi_k \rangle_{\mathsf{obj}}$ and implement the proof checker using a formal language called Metamath [25].

4.1.1 Metamath Overview

Metamath is a simple formal language that can define the meta-theory of a logical system and prove meta-theorems about it, accompanied by proofs that can be verified by a computer program, called a Metamath proof verifier. At its core, Metamath is a substitution system. The main data that Metamath manipulates is that of a sequence of Metamath tokens, which are either Metamath constants or Metamath variables.[2] A Metamath variable can be substituted for a sequence of Metamath tokens that may include both Metamath variables or Metamath constants. Metamath allows users to define certain Metamath token sequences as Metamath axioms, from which, by substitution, more Metamath token sequences can be obtained, i.e. "proved", as Metamath theorems. A Metamath proof of a Metamath theorem is essentially an encoding of the substitution operations that have been applied in order to obtain the proved theorem. Although Metamath also has a few other features, such as supporting sorts and allowing to declare two Metamath variables as "distinct" (whose formal meaning is not important here), what was described above covers the core functionality of Metamath.

A Metamath proof verifier takes a Metamath theorem and its proof and applies the substitution operations specified in then proof to see if the result is the same as the theorem. The main advantage of Metamath, which is also the main reason I decide to implement the first prototype of the matching logic proof checker in it, is its simplicity and efficiency. This is explained later. Before that, I explain how Metamath is used to define a logic L and to prove the meta-theorems of L.

The way how Metamath is used to define a logic L and to prove the meta-theorems of L is explained below. A Metamath definition that consists of the following basic components is prepared:

1. A Metamath definition of L:
   (a) a declaration of some Metamath constants that are used to represent the logical constructs of L, the syntactic categories in the syntax of L (such as that of the formulas of L) as the sorts in Metamath, and some meta-level syntax that is needed to state the meta-theorems. A common one is the proof entailment symbol "$\vdash$";
   (b) a declaration of some Metamath variables that are used to represent the meta-variables of L;
   (c) a set of Metamath axioms that define the syntax and proof system of L;
2. a set of Metamath theorems, accompanied by their Metamath proofs, which prove the given meta-theorems of L.

Intuitively, item 1 forms a specification of a proof checker of L and item 2 forms a proof object of the given meta-theorems. The proof checking function $\mathsf{ProofCheck}$ of L is the Metamath proof checking algorithm

Metamath has been successfully applied to formalize many important mathematical logics and/or theories. The most substantial work that has been done using Metamath is a 40-million-LOC formalization of the ZFC set theory axioms as well as the basic and more advanced mathematics that is defined based on set theory, accompanied by 23,000 completely worked out Metamath proofs.

[2] Although verbose, we keep the premodifier "Metamath" so as not to confuse with the concepts in matching logic.

Metamath is known for its simplicity and efficiency. Since I am pursuing a similar goal with the matching logic proof checker, I decide to implement the first prototype in Metamath so as to take advantage of the state of the art tools. There are more than 15 Metamath proof verifiers, all compact and short. In terms of the size, the smallest verifier is written in 74 lines of Mathematica code. Besides that, a Python verifier has 350 lines of code, a Lua one has 380 lines of code, and a Haskell one has 400 lines of code. One of the fastest Metamath verifier is written in C^{++}, which can complete the verification of the aforementioned Metamath definition of set theory with 23,000 Metamath proofs in fewer than 20 seconds.

Next, I describe the Metamath definition of matching logic.

4.1.2 Matching Logic Proof Checker Implemented in Metamath

I briefly discuss the design of a 245-LOC prototype matching logic proof checker [23] written in Metamath.

Firstly, I define the primitive syntax of matching logic as Metamath constants.

```
  $c \bot \imp \app \ex \mu ( ) $.
```

Here, the Metamath notation "`$c ... $.`" is used to declare a list of space-separated constants. I declare the bottom `\bot`, implication `\imp`, application `\app`, the existential quantification `\ex`, and the least fixpoint operator `\mu`. I also define the parentheses, `(` and `)`, to group the syntax.

Secondly, I define the following syntactic categories of matching logic using Metamath constants.

```
  $c #Pattern #ElementVariable #SetVariable #Symbol #Variable $.
```

For each syntactic category in the above, I define the meta-variables that range over it. For example, I define `ps` and `ph` as the meta-variables for matching logic patterns as follows:

```
  vph  $f #Pattern ph $.
  vps  $f #Pattern ps $.
```

Here, the Metamath notation "`$f ... $.`" is used to declare a meta-variable of a given syntactic category. The identifiers `vph` and `vps` are given to the meta-variable declarations so that the other proofs can refer to them.

Thirdly, I define some auxiliary predicates on the syntax of matching logic, such as checking the free occurrences of a variable in a pattern, which are are necessary because they appear in the side conditions of matching logic proof rules. Some of them are also used in defining the pattern syntax, as shown below.

Fourthly, I define matching logic syntax. In Metamath, this means to define which sequences of the Metamath constants and/or variables are regarded as members of the syntactic category `#Pattern`. The following is the Metamath definition of matching logic syntax.

```
 wv $a #Pattern xX $.
 wb $a #Pattern \bot $.
 wi $a #Pattern ( \imp ph ps ) $.
 wa $a #Pattern ( \app ph ps ) $.
 we $a #Pattern ( \ex x ph ) $.
 ${ wm.1 $e #NoNegativeOccurrence X ph $.
    wm   $a #Pattern ( \mu X ph ) $.
 $}
```

Here, `xX` is a declared meta-variable that ranges over both the element and the set variables.[3]

[3] For simplicity, I do not show the declaration here.

Similarly, `x` and `X` are declared as meta-variables that range over the element variables and set variables, respectively. In the above definition, I adopt an S-expression style syntax to encode matching logic patterns in Metamath. Note that in the definition of the syntax of $\mu X \,.\, \varphi$, it is required that X has no positive occurrences in φ.

Fifthly, I define matching logic proof system. It includes the following definition of the entailment symbol:

```
$c |- $.
```

and Metamath axioms that define the matching logic proof rules. For example, the following is the Metamath definition of the (Modus Ponens) proof rule:

```
${  pr-mp.1   $e |- ps $.
    pr-mp.2   $e |- ( \imp ps ph ) $.
    pr-mp     $a |- ph $.
$}
```

Finally, I define an equality relation at the meta-level of matching logic so that syntactic sugar can be defined and their properties can be formally proved from the definitions (instead of axiomatized).

The resulting Metamath definition is a file ml.mm with 245 lines of Metamath code that fully implements the meta-theory of matching logic. Using this Metamath definition, I can write proof objects as Metamath theorems accompanied with their Metamath proofs. For example, the following Metamath theorem states that the pattern $\varphi \to \varphi$ is provable for any pattern φ:

```
iid $p |- ( \imp ph ph ) $=
  vph vph wi vph vph vph wi wi vph vph pr-1
  vph vph vph wi wi vph vph wi wi vph vph
  vph wi vph wi wi vph vph vph wi pr-1 vph
  vph vph wi vph pr-2 pr-mp pr-mp $.
```

In the above, the Metamath notation "`$p ... $= ... $.`" is used to state a theorem and its proof.

4.1.3 Generating Proof Objects for Program Execution and Verification

In matching logic, the formal semantics of any programming language L is given as a set of reachability rules. In Section 3.2, I show that reachability rules can be defined as matching logic patterns. Therefore, the formal semantics of L is given by a set of matching logic axioms. Let Γ^L denote the resulting matching logic theory that includes the formal semantics of L.

In Section 3.2 and [6], I also recall that the dynamic properties of programs such as execution and verification can be expressed by patterns. For example, that the initial configuration φ_0 executes to the final configuration φ_{100} in 100 steps, or that the pre-condition φ_{pre} eventually reaches the post-condition φ_{post} (in a partial-correctness manner), can be expressed by the following matching logic patterns, respectively:

$$\Gamma^L \vdash \varphi_0 \Rightarrow^{100} \varphi_{100} \qquad\qquad \Gamma^L \vdash \varphi_{pre} \Rightarrow \varphi_{post}$$

To encode the above as proof objects, I need to translate the axioms in Γ^L into Metamath axioms that state that each pattern in Γ^L is provable. The above properties are then stated as Metamath theorems.

4.2 Matching Logic Complete Deduction

From Section 3.4 we know that matching logic has no complete proof system. If no proof system is complete, how do we know the current proof system in Figure 2 is "good enough" and not merely a set of ad-hoc proof rules? To answer this question, we need to establish a clear and elegant relationship between matching logic semantics and matching logic proof system.

Theorems 1 and 2 are the completeness results that have been proved for matching logic. They show that the proof rules in Figure 2 that handle non-fixpoint reasoning are not ad-hoc because they are complete with respect to the empty theory and theories where equality is present. Therefore, the main technical difficulty is the completeness with respect to fixpoints.

I propose the following method based on *Henkin semantics*, also known as *general semantics*, which gives fixpoint patterns an alternative semantics. In the current semantics, called *standard semantics*, the least fixpoint pattern $\mu X \, . \, \varphi$ is required to be interpreted as the smallest set X such that the equation $X = \varphi$ holds (note that φ may include X). However, in the proposed Henkin semantics, the least fixpoint $\mu X \, . \, \varphi$ is not necessarily interpreted as the true least fixpoint in the models. Instead, each model M is extended by a nonempty domain $\mathcal{D} \subseteq \mathcal{P}(M)$ as the range of all possible interpretation of the fixpoint patterns. We call each such pair $\langle M, \mathcal{D} \rangle$ a *Henkin model*. Least fixpoint pattern $\mu X \, . \, \varphi$ is then interpreted as the smallest set that belongs to $\mathcal{D}$ such that $X = \varphi$. When $\mathcal{D} = \mathcal{P}(M)$ is the full power set of M, the Henkin model $\langle M, \mathcal{D} \rangle$ yields the standard semantics. When $\mathcal{D} \neq \mathcal{P}(M)$, Henkin semantics allows to have *nonstandard models* where least fixpoints are not interpreted as the true least fixpoints in the models but only the least ones in $\mathcal{D}$.

The above Henkin extension of the standard semantics is classical and has been successfully applied to second-order logic as well as first-order modal μ-calculus and has obtained the corresponding completeness results.

4.3 Matching Logic Decidable Fragments

Decision procedures play an important role in automated reasoning. Matching logic is not decidable, but it has fragments that are decidable. The simplest fragment is *propositional logic fragment* that consists of set variables (serving as propositional variables) and the two propositional connectives $\bot$ and $\varphi_1 \to \varphi_2$, and no other constructs. However, no further research has been carried out on matching logic decidability, which has caused $\mathbb{K}$ to utilize the existing decision procedures such as SMT solvers in an ad-hoc way.

In the proposed research, I will systematically study decidable fragments of matching logic and implement the corresponding decision procedures. These decision procedures can be integrated into $\mathbb{K}$ and improve the level automation in the current $\mathbb{K}$ implementations.

Satisfiability. Firstly, I define the decision problem of matching logic satisfiability modulo theory, abbreviated MLSMT, as follows:

> Given a theory Γ and a pattern φ, is there a model M and a valuation ρ of variables such that $M \vDash \Gamma$ and $[\![\varphi]\!]_{M,\rho} \neq \emptyset$, where $[\![\varphi]\!]_{M,\rho}$ is the interpretation of φ in M under ρ?

MLSMT is an important decision problem. It can be used to encode many other interesting and important decision problems. For example, it can be used to encode the following *validity problem*:

> Given a theory Γ and a pattern φ, whether $\Gamma \vDash \varphi$?

Indeed, validity can be solved by calling MLSMT with Γ and $\neg\varphi$. If $\neg\varphi$ is not satisfiable (modulo Γ) then φ is valid, and vice versa.

Another application of MLSMT is to check the *consistency* of a logical theory Γ in the following sense. We call MLSMT with Γ and $\top$ (called "top", defined by $\top \equiv \neg\bot$). If it returns that $\top$ is satisfiable under Γ, we know that Γ is not inconsistent because it has a satisfying model.

Finite/small model property. Finite model property is closely related to satisfiability. It studies the *size* of the satisfying model M of pattern φ. Formally, we say that the finite model property holds if for any φ that is satisfiable modulo a theory Γ, there exists a satisfying model M whose size is *finite*. If additionally, the size of M is bounded by $O(f(|\varphi|))$ where f is a computable function and $|\varphi|$ is the size of φ, we say that the *small model property* holds. The known results and open problems on small model property are summarized in Table 1.

Finite/small model property is often a byproduct of decidability. As I show later, decision procedures of MLSMT are based on the tableau method and build at runtime a satisfying model for the given input pattern φ using tableau rules. The built models have a tree-like structure, possibly infinite, but can be folded into a finite graph-like structure and thus fulfill the finite/small model property.

Syntactic fragments. Generally speaking, MLSMT is undecidable for matching logic (see Table 1). Therefore, I propose to study the *decidable fragments* of matching logic. Recall that matching logic has 8 syntactic constructs (Section 3.1). Except $\bot$ and $\varphi_1 \to \varphi_2$ that build propositional constraints as well as user-provided symbols and application, I regard the remaining 4 constructs as optional and denote them using the following tokens:

1. ev: element variables;
2. sv: set variables;
3. exists: the $\exists$ quantifier; and
4. mu: the μ operator.

A *syntactic fragment* can be described by a subset $\Pi \subseteq \{\mathsf{ev}, \mathsf{sv}, \mathsf{exists}, \mathsf{mu}\}$ such that

1. $\mathsf{exists} \in \Pi$ implies $\mathsf{ev} \in \Pi$, and
2. $\mathsf{mu} \in \Pi$ implies $\mathsf{sv} \in \Pi$.

Under the above classification, there are $3 \times 3 = 9$ different syntactic fragments, where the first 3 denotes a choice in $\{\emptyset, \{\mathsf{ev}\}, \{\mathsf{ev}, \mathsf{exists}\}\}$ and the second 3 denotes a choice in $\{\emptyset, \{\mathsf{sv}\}, \{\mathsf{sv}, \mathsf{mu}\}\}$. In Table 1, four syntactic fragments are shown.

Decidability of fragments involving fixpoints. It is especially interesting to study decidability when fixpoints are present. Therefore, I start by studying the syntactic fragment $\Pi_\mu = \{\mathsf{sv}, \mathsf{mu}\}$, that is, the fragment of matching logic that excludes element variables or the quantification $\exists$, but allows set variables and least fixpoints. Recall that the greatest fixpoints can be defined from least fixpoints and negation. I also assume that the given theory $\Gamma = \emptyset$. Nonempty theories are to be considered in the future, too.

$$(\textsc{and})\qquad \frac{\varphi_1 \wedge \varphi_2, \Gamma}{\varphi_1, \varphi_2, \Gamma}$$

$$(\textsc{or-l})\qquad \frac{\varphi_1 \vee \varphi_2, \Gamma}{\varphi_1, \Gamma}$$

$$(\textsc{or-r})\qquad \frac{\varphi_1 \vee \varphi_2, \Gamma}{\varphi_2, \Gamma}$$

$$(\textsc{ons})\qquad \frac{U, \Gamma}{\varphi[U/X], \Gamma} \qquad \text{where } U = \kappa X \,.\, \varphi \text{ and } \kappa \in \{\mu, \nu\}$$

$$(\textsc{mu})\qquad \frac{\mu X \,.\, \varphi, \Gamma}{U, \Gamma} \qquad \text{where } U = \mu X \,.\, \varphi$$

$$(\textsc{nu})\qquad \frac{\nu X \,.\, \varphi, \Gamma}{U, \Gamma} \qquad \text{where } U = \nu X \,.\, \varphi$$

$$(\textsc{app-1})\qquad \frac{\Gamma}{\{\Gamma \rightsquigarrow \mathit{app}(\varphi_1, \varphi_2) \mid \mathit{app}(\varphi_1, \varphi_2) \in \Gamma\}}$$

$$(\textsc{app-2})\qquad \frac{\Gamma \rightsquigarrow \mathit{app}(\varphi_1, \varphi_2)}{\Gamma \rightsquigarrow \mathit{app}(\varphi_1, \varphi_2) \rightsquigarrow (\Gamma_{\overline{\mathit{app}},1}, \Gamma_{\overline{\mathit{app}},2})}$$

$$(\textsc{app-3})\qquad \frac{\Gamma \rightsquigarrow \mathit{app}(\varphi_1, \varphi_2) \rightsquigarrow (\Gamma_{\overline{\mathit{app}},1}, \Gamma_{\overline{\mathit{app}},2})}{\{\varphi_i, \Gamma_{\overline{\mathit{app}},i} \mid i \in \{1, 2\}\}}$$

Figure 5: The proposed tableau rules for the Π_μ-fragment that yield decidability.

Π_μ-fragment can be regarded as an extension of modal μ-calculus, whose syntax also defines propositional connectives, set variables (called propositional variables or atomic propositions) and the least fixpoint operator μ. The only difference is that modal μ-calculus defines formulas of the form $\circ\varphi$ while matching logic defines a binary application construct $\varphi_1\,\varphi_2$. This difference can be resolved by defining $\circ$ as a symbol in matching logic and regarding $\circ\varphi$ as the matching logic symbol/pattern $\circ$ applied to φ. This way, modal μ-calculus becomes a fragment of matching logic Π_μ-fragment.

Note that modal μ-calculus is decidable. The close relationship between modal μ-calculus and the Π_μ-fragment of matching logic implies that the proof techniques of the decidability of modal μ-calculus, called the *tableau method* [27], may be applied to the Π_μ-fragment. In the tableau method, a set of tableau rules is used to build a satisfying model for φ. By construction, the model built by the tableau method has a tree-like structure, where each node is associated with a set of patterns that are sub-patterns of φ and are satisfied by the node. This invariant is maintained throughout the construction and is used to prove that the resulting model is indeed a satisfying model of φ. In Figure 5, I list the proposed tableau rules. I have used the proposed tableau rules to prove following decidability result of matching logic, which is the first decidability result for matching logic of its kind:

Theorem 3 *MLSMT is decidable for the empty theory $\emptyset$ on the Π_μ-fragment.*

4.4 Defining Logical Systems in Matching Logic

Apart from the logical systems in Figure 3, there are still many important logical systems that have not been defined as matching logic theories.

In this proposed work, I plan to consider two logical systems and define them in matching logic. They are separation logic and hyper linear temporal logic (hyperLTL).

Separation logic. Separation logic [30] was initially proposed to specify and reason about data structures on heaps and has developed into a family of logics that deal with resources in a computing system. For example, Iris [22] is a framework where resources and their ownership can be formalized.

The vanilla separation logic as proposed in the original paper [30] uses the following syntax to build *heap assertions*:

$$\varphi ::= \text{(FOL syntax)} \mid \mathit{emp} \mid l \mapsto v \mid \varphi_1 * \varphi_2 \mid \varphi_1 \mathrel{-\!\!*} \varphi_2$$

Semantically, a heap is a finite mapping $h \colon L \rightharpoonup V$ where L is a set of locations and V is a set of values. There is a distinguished location $\mathit{nil} \in L$ that denotes the nil location. Any heap h must be undefined on nil. Empty heap $\bot_{\mathit{heap}}$ is undefined everywhere.

Given a valuation ρ of the variables appearing in a heap assertion φ, we define the semantic relation $h, \rho \vDash_{\mathsf{SL}} \varphi$ to mean that h satisfies φ under ρ. In the syntax of separation logic, $\varphi_1 * \varphi_2$ is called separating conjunction and $\varphi_1 \mathrel{-\!\!*} \varphi_2$ is called separating implication. These are the two most important constructs in separation logic. The semantics of separating conjunction states that any heap h such that $h, \rho \vDash_{\mathsf{SL}} \varphi_1 * \varphi_2$ can be divided into two disjoint heaps, written h_1 and h_2, such that $h = h_1 \cup h_2$ and $h_i, \rho \vDash_{\mathsf{SL}} \varphi_i$ for $i \in \{1, 2\}$. The semantics of separating implication is the reverse of separating conjunction. Any heap h such that $h, \rho \vDash_{\mathsf{SL}} \varphi_1 \mathrel{-\!\!*} \varphi_2$ satisfies the following property: if extended with a disjoint heap h_1 such that $h_1, \rho \vDash_{\mathsf{SL}} \varphi_1$, the resulting heap $h \cup h_1$ satisfies $h \cup h_1, \rho \vDash_{\mathsf{SL}} \varphi_2$.

In [31], it is shown that (vanilla) separation logic is a special instance of matching logic, if we fix the underlying (matching logic) model to be the *standard model of maps*, written *Map*. Specifically, we can define the constructs of heap assertions, emp, $l \mapsto v$, and $\varphi_1 * \varphi_2$, as symbols/constructors in matching logic. Then we consider the standard model of (finite) maps *Map* from L to V. Model *Map* has elements that are finite maps from L to V and interpret the heap constructs in the standard way. Then, it is proved that heap assertions have the same semantics under separation logic and in the matching logic model *Map*, in the following sense:

$$h, \rho \vDash_{\mathsf{SL}} \varphi \qquad \text{iff} \qquad h \in [\![\varphi]\!]_{\mathit{Map},\rho}$$

What is missing above and proposed as future work is an *axiomatization* of the standard map model *Map*, without which we cannot phrase the above equivalence as one that is between separation logic and a matching logic theory Γ^{Map}, which is necessary if we are to generate proof objects for heap-manipulating programs whose correctness properties are expressed in separation logic heap assertions.

In the proposed research, I will define the theory Γ^{Map} that completely captures the standard map model *Map*. The key is to capture the inductive principle of the set of finite maps using the least fixpoint operator μ by the following pattern/axiom:

$$\mu M \,.\, \mathit{emp} \vee (\exists l \,.\, \exists v \,.\, l \mapsto v) \vee (M * M)$$

Intuitively, the above axiom states that the carrier set is the smallest set M that is closed under emp, $l \mapsto v$, and $\varphi_1 * \varphi_2$. Separating implication can be handled following the same method proposed in [31], in the following way:

$$\varphi_1 \mathrel{-\!\!*} \varphi_2 \equiv \exists h{:}\mathit{Heap} \, . \, h \wedge h * \varphi_1 \subseteq \varphi_2$$

Intuitively, the above pattern $\varphi_1 \mathrel{-\!\!*} \varphi_2$ is matched by the heaps h such that any heaps matching $h * \varphi_1$ also match φ_2, thus yielding the same semantics as separation implication.

Hyperproperties. Hyperproperties were firstly introduced in [11] to specify the relationship among multiple execution traces. They are often used to specify security policies such as noninterference, noninference, and declassification. Many logics have been proposed to specify and reason about hyperproperties, such as hyperLTL and hyperCTL* [10], but none of them have been defined in matching logic.

To enable 𝕂's ability to specify and reason about hyperproperties, I propose to study hyperLTL and define it in matching logic as a logical theory. In short, hyperLTL extends the classical linear temporal logic (LTL) by *trace quantifiers*, written $\exists\pi$ and $\forall\pi$, that are used to specify that the properties hold on some/all traces. For example, $\forall\pi_1 \, . \, \exists\pi_2 \, . \, \varphi$ means that for all traces π_1 there exists a trace π_2 such that φ on these two traces. Additionally, hyperLTL introduces *atomic trace propositions* written a_π, which hold if the atomic proposition a holds on π in the current state. This way, one trace can check whether some propositions hold on the other traces.

In [6], it has been proved that linear temporal logic (LTL) can be defined in matching logic, where $\circ\varphi$ (read "next φ") is defined by applying the (matching logic) symbol $\circ$ to φ. The other modal operators such as $\diamond\varphi$ (read "eventually φ"), $\Box\varphi$ (read "always φ"), and $\varphi_1 \, U \, \varphi_2$ (read "φ_1 until φ_2) can be defined by the following matching logic patterns:

$$\begin{aligned}
&\diamond \varphi \equiv \mu X \, . \, \varphi \vee \bullet X \\
&\Box\varphi \equiv \nu X \, . \, \varphi \wedge \bullet X \\
&\varphi_1 \, U \, \varphi_2 \equiv \mu X \, . \, \varphi_2 \vee (\varphi_1 \wedge \bullet X)
\end{aligned}$$

The matching logic theory that defines hyperLTL is obtained by extending the above definition of LTL by trace quantifiers and atomic trace propositions, in the following way:

$$\forall\pi \, . \, \varphi \equiv \forall\pi{:}\mathit{InitState} \, . \, \varphi$$

where *InitState* is the sort of the initial states, from which the traces that are quantified by the trace quantifiers are generated.

5 Conclusion

The goal of my research is to demonstrate a trustworthy and practical language framework. My research is based on the 𝕂 framework that has been shown successful in formalizing large programming languages in practice. My main methodology is to define a logical foundation for 𝕂 using matching logic. Using matching logic, the formal semantics of a language given in 𝕂 becomes a logical theory. The correctness of 𝕂 is established on a case-by-case basis for each individual tasks that 𝕂 conducts. My research not only studies the fundamental problems about matching logic such as its expressiveness, complete deduction, and decidable fragments, but also covers the implementation of an automated theorem prover for matching logic and an efficient proof checker.

References

[1] Egidio Astesiano, Michel Bidoit, Hélène Kirchner, Bernd Krieg-Brückner, Peter D. Mosses, Donald Sannella, and Andrzej Tarlecki. CASL: the common algebraic specification language. *Journal of Theoretical Computer Science*, 286(2):153–196, 2002. Current trends in Algebraic Development Techniques.

[2] Henk Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, volume 2, background: computational structures, chapter 2, pages 117–309. Oxford University Press, UK, 1993.

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

[4] Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system F with subtyping. *Information and Computation*, 109(1):4–56, 1994.

[5] 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.

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

[8] Xiaohong Chen, Minh-Thai Trinh, Nishant Rodrigues, Lucas Peña, and Grigore Roşu. Towards a unified proof framework for automated fixpoint reasoning using matching logic. In *PACMPL Issue OOPSLA 2020*, pages 1–29. ACM/IEEE, November 2020.

[9] Alonzo Church. *The calculi of lambda-conversion*. Princeton University Press, Princeton, New Jersey, USA, 1941.

[10] Michael R Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K Micinski, Markus N Rabe, and César Sánchez. Temporal logics for hyperproperties. In *International Conference on Principles of Security and Trust*, pages 265–284. Springer, 2014.

[11] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, 2010.

[12] 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.

[13] 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.

[14] Razvan Diaconescu and Kokichi Futatsugi. *CafeOBJ report: the language, proof techniques, and methodologies for object-oriented algebraic specification*, volume 6 of *AMAST Series in Computing.* World Scientific, Singapore, 1998.

[15] Jean-Yves Girard. The system F of variable types, fifteen years later. *Theoretical Computer Science*, 45:159–192, 1986.

[16] Kurt Gödel. *On formally undecidable propositions of principia Mathematica and related systems.* Courier corporation, 1992.

[17] Joseph Goguen, James Thatcher, Eric Wagner, and Jesse Wright. Initial algebra semantics and continuous algebras. *Journal of the ACM*, 24(1):68–95, 1977.

[18] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. *Software engineering with OBJ: Algebraic specification in action*, chapter Introducing OBJ, pages 3–167. Springer, Massachusetts, USA, 2000.

[19] Dwight Guth. A formal semantics of Python 3.3. Master's thesis, University of Illinois at Urbana-Champaign, August 2013.

[20] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. Defining the undefinedness of C. In *Proceedings of the 36th annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*, pages 336–345, Portland, OR, 2015. ACM.

[21] 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`.

[22] Ralf Jung, Robbert Krebbers, Jacques-henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018.

[23] K Team. Matching logic proof checker. GitHub page `https://github.com/kframework/matching-logic-prover/tree/master/checker`, 2020.

[24] James McKinna and Robert Pollack. Pure type systems formalized. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications*, pages 289–305, Berlin, Heidelberg, 1993. Springer.

[25] Norman D. Megill and David A. Wheeler. *Metamath: a computer language for mathematical proofs.* Lulu Press, Morrisville, North Carolina, 2019. `http://us.metamath.org/downloads/metamath.pdf`.

[26] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes (part 1). *Information and Computation*, 100(1):1–40, 1992.

[27] Damian Niwiński and Igor Walukiewicz. Games for the μ-calculus. *Theoretical Computer Science*, 163(1):99–116, 1996.

[28] 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.

[29] Andrew Pitts. Construction of the initial algebra for a strictly positiveendofunctor on Set using uniqueness of identity proofs, functionextensionality, quotients types and sized types. Available at `www.cl.cam.ac.uk/users/amp12/agda/initial-T-algebras`., November 2019.

[30] John C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS'02)*, pages 55–74, Copenhagen, Denmark, 2002. IEEE.

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

[32] L. S. van Benthem Jutting. Typing in pure type systems. *Information and Computation*, 105(1):30–41, 1993.

[33] Mark van den Brand, Arie van Deursen, Jan Heering, Hayco de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A. Olivier, Jeroen Scheerder, Jurgen J. Vinju, Eelco Visser, and Joost Visser. The ASF+SDF meta-environment: a component-based language development environment. *Electronic Notes in Theoretical Computer Science*, 44(2):3–8, 2001.