MATCHING μ-LOGIC

Draft of May 2, 2023 at 09 : 31

BY

XIAOHONG CHEN

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2023

Urbana, Illinois

Doctoral Committee:

Professor Grigore Roşu, Chair
Professor José Meseguer
Professor Madhusudan Parthasarathy
Doctor Margus Veanes, Microsoft Research

Abstract

We present matching μ-logic, which is a unifying logic for specifying and reasoning about programs and programming languages. Matching μ-logic uses its formulas, called patterns, to uniformly express programs' static structures, dynamic behaviors, and logical constraints. Programming languages can be formally defined as matching μ-logic theories, which include patterns as axioms. The correctness of programming language implementations and tools can be proved using a fixed proof system. These proofs can be encoded as proof objects and automatically checked using a small proof checker.

An important feature of matching μ-logic is its μ operator, which provides direct support for specifying fixpoints and thus enables to specify and reason about induction and recursion.

We study the proof theory of matching μ-logic and prove a few important completeness results. We study the expressive power of matching μ-logic and show that many important logics, calculi, and foundations of computations, especially those featuring fixpoints/induction/recursion, can be defined as matching μ-logic theories.

We study automated reasoning for matching μ-logic, with a focus on fixpoint reasoning. We propose a set of high-level automated proof rules that can be applied to many matching μ-logic theories, and thus enable automated reasoning in them.

We propose applicative matching μ-logic, abbreviated as AML, as a simple instance of matching μ-logic that remains all of its expressive power. AML is obtained by eliminating the features of matching μ-logic that are definable using the other more basic and primitive infrastructure. We present an encoding of matching μ-logic into AML and implement a 240-line proof checker for AML using Metamath.

We study proof-certifying program execution and formal verification, where the correctness of an execution/verification task is established by an AML proof object, serving as the machine-checkable correctness certificate. Our approach is based on the $\mathbb{K}$ formal language semantics framework. We design and implement procedures that output AML proof objects for the language-agnostic program interpreter and formal verifier of $\mathbb{K}$, which are parametric in the formal semantics of a programming language. This way, the correctness of program execution or formal verification is reduced to checking the corresponding AML proof objects using the proof checker.

We hope to demonstrate that matching μ-logic can serve as a unifying foundation for programming, where programming languages are defined as theories, and the correctness of language implementations and tools is established by machine-checkable proof objects.

To my parents.

Acknowledgments

I express my sincere gratitude to my advisor Grigore Roşu for his continuous support during my studies. I thank my thesis committee for their insightful comments and valuable feedback. I thank all my collaborators and fellow doctoral students. And finally, I want to give my deepest appreciation to my parents, who assisted me through this long journey with advice and love.

Table of Contents

Chapter 1: INTRODUCTION

Unlike natural languages that allow vagueness and ambiguity, programming languages must be precise and unambiguous. Only with rigorous definitions of programming languages, called their formal semantics, can we guarantee the reliability, safety, and security of computing systems. Our vision is thus a unifying programming language framework based on the formal semantics of programming languages, as shown in Figure 1.1. In such an ideal language framework, language designers only need to define the formal semantics of their languages. All the implementations and tools of a given programming language are automatically generated from the formal semantics of the language by the framework.

A unifying language framework requires a unifying logical foundation, where the formal semantics of programming languages are defined as logical theories, consisting of logical formulas as axioms that specify the behaviors of programs. The correctness of language implementations and tools can be proved using a fixed formal proof system. These formal proofs can be encoded as proof objects and checked by a proof checker. A unifying language framework with a unifying logical foundation allows us to reduce the correctness of computation and programming in general to something as simple as proof checking.

Previous work has pursued the above vision with the $\mathbb{K}$ framework [1] and matching logic [2]. $\mathbb{K}$ is a rewrite-based language framework that allows to define the formal semantics of programming languages using configurations and rewrite rules. From the formal semantics of any given programming language, $\mathbb{K}$ automatically generates a set of language tools, including a parser, a program interpreter, a formal verifier, and a program equivalence checker [3, 4]. $\mathbb{K}$ has been used to define the complete executable formal semantics of many large languages, such as C [5], Java [6], JavaScript [7], Python [8], Ethereum virtual machines bytecode [9], and x86-64 [10]. Matching logic has served as the logical foundation for the static aspects of $\mathbb{K}$. The core of matching logic is a notion of its formulas called *patterns*, which can be used to uniformly specify and reason about program configurations and logical constraints.

However, matching logic has two major limitations that prevent it from being able to serve as the unifying logical foundation for programming. The first limitation is the lack of a universal proof system that supports formal reasoning in all matching logic theories. The known matching logic proof system $\mathcal{P}$ proposed in [2] is not universal because it only supports formal reasoning in a subset of theories that feature *definedness*—a mathematical instrument that can be used to define equality. If the underlying theory does not feature definedness, $\mathcal{P}$ cannot be used to do formal reasoning in it.

The second limitation of matching μ-logic is the lack of ability to specify and reason about

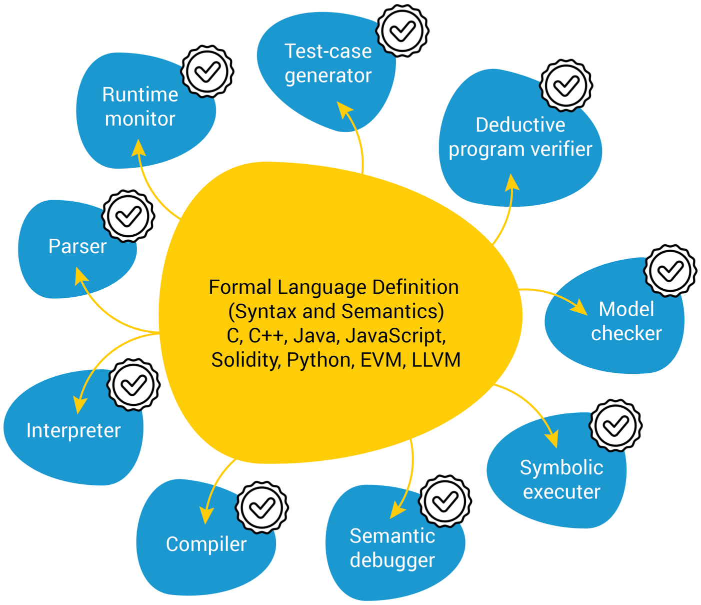


Figure 1.1: Vision of a Unifying Programming Language Framework

fixpoints. Fixpoints are ubiquitous and unavoidable in computer science. Without a direct support for fixpoints, matching logic is insufficient for dealing with topics such as inductive datatypes, induction principles, temporal properties about programs, or formal verification. To handle these fixpoints-related topics, one has to defer them outside matching logic to some other logics or frameworks such as Coq [116], or extend matching logic with additional infrastructure for fixpoints. For example, matching logic has been extended to reachability logic [11] that provides additional coinduction-based proof rules for formal verification.

This work addresses the above two limitations of matching logic. For the first limitation, we propose a new proof system $\mathcal{H}$ for matching logic that is universal and works with all theories. We show that $\mathcal{H}$ is sound for all theories, meaning that if a pattern (i.e., a matching logic formula) is provable using $\mathcal{H}$ in a given theory, then it holds in that theory. The other direction is known as the completeness of $\mathcal{H}$. While we do not know whether $\mathcal{H}$ is complete for all theories, we present two important completeness results. The first is the definedness completeness theorem (Theorem 3.8) which states that $\mathcal{H}$ is complete for every theory that features definedness. The second is the local completeness theorem (Theorem 3.14), which states that $\mathcal{H}$ is complete for the empty theory. We present proof system $\mathcal{H}$ in Chapter 3.

For the second limitation, we extend matching logic to matching μ-logic, by adding a μ operator that builds least fixpoints. Greatest fixpoints are definable using least fixpoints. We also extend the proof system $\mathcal{H}$ to $\mathcal{H}_\mu$, which has two proof rules dedicated to fixpoint reasoning, inspired from the Knaster-Tarski fixpoint theorem (Theorem 2.1). This way, matching μ-logic can specify and reason about fixpoints in a principled way. We present matching μ-logic and its proof system $\mathcal{H}_\mu$ in Chapter 4.

We then proceed to study the expressive power of matching μ-logic. We show that many important logics, calculi, and foundations of computations, especially those featuring fixpoints, can be defined as matching μ-logic theories. These includes FOL with least fixpoints, initial algebra semantics, separation logic with recursive predicates, modal μ-calculus, various temporal logics, dynamic logic, reachability logic, λ-calculus, and type systems. We present the above results about the expressive power of matching μ-logic in Chapter 5.

We study automated reasoning for matching μ-logic with a focus on fixpoint reasoning. We propose a unifying proof framework that consists of high-level proof rules that are derivable using the proof system $\mathcal{H}_\mu$. Automated reasoning becomes proof search over the proposed high-level proof rules, with heuristics guiding the proof search for better performance. We present the above unifying proof framework based on matching μ-logic in Chapter 6.

We propose and study applicative matching μ-logic, abbreviated as AML, which is a simple instance of matching μ-logic that remains all of its expressive power. AML is obtained by eliminating the features of matching μ-logic that are definable using the other more basic and primitive infrastructure. For example, sorts in matching μ-logic are definable so they are eliminated in AML. We present an encoding of matching μ-logic into AML. We implement a proof checker for AML using Metamath [107] in 240 lines of code. AML proofs can be encoded as proof objects and checked by the 240-line proof checker, which serves as a small trust base of checking any AML proofs. We present AML and the proof checker in Chapter 7.

Finally, we put everything together and study proof-certifying program execution and formal verification, based on the $\mathbb{K}$ framework. We implement proof generation procedures for the program interpreter and the formal verifier of $\mathbb{K}$, which are parametric in the formal semantics of a programming language. The proof generation procedures output AML proof objects as correctness certificates for the said interpreter and verifier. This way, we reduce the correctness of program execution and formal verification to checking the corresponding AML proof objects. We discuss proof-certifying program execution in Chapter 8 and proof-certifying formal verification in Chapter 9.

The vision of a unifying language framework and a unifying logic foundation for programming is a grand one. Related study started in the 1960s, with the proposal of various formal semantics notions and styles [12, 13, 14, 15, 16, 17, 18]. After more than half a century of research on the topic, great progress has been made in terms of the scalability, usability, robustness, popularity, reusability, and trustworthiness of semantics-based language tools, moving us closer to realizing the above vision, which we believe, with evidence present in this work, is in within our reach in the near future with matching μ-logic.

Chapter 2: PRELIMINARIES

This preliminary chapter consists of three parts. The first part is Section 2.1, where we review the basic definitions and notation in mathematics, such as sets, functions, and relations. The second part is Sections 2.2–2.14, where we introduce the logics, calculi, and foundations of computation that are relevant to this work. The third part is Section 2.15, where we present the $\mathbb{K}$ formal language semantics framework.

2.1 BASIC MATHEMATICS

Let A be a set. The size or *cardinality* of A is denoted by $\mathsf{card}(A)$. The *powerset* of A is denoted by $\mathcal{P}(A)$. The empty set is denoted by $\emptyset$.

Let A and B be two sets. The *intersection* of A and B is denoted by $A \cap B$. The *union* of A and B is denoted by $A \cup B$. The *set difference* of A and B is denoted by $A \setminus B$ and defined by $A \setminus B = \{a \in A \mid a \notin B\}$. The *set symmetric difference* of A and B is denoted by $A \bigtriangleup B$ and defined by $A \bigtriangleup B = (A \setminus B) \cup (B \setminus A)$. If $A \cap B = \emptyset$, we say that A and B are *disjoint*. We write $A \dot{\cup} B$ to mean $A \cup B$, with the assumption that A and B are disjoint. We write $A \subseteq B$ to mean that A is a *subset* of B. We write $A \subsetneq B$ to mean that A is a *strict subset* of B, that is, $A \subseteq B$ and $A \neq B$.

A *total function* or simply *function* from A to B is denoted by $f \colon A \to B$, whose *domain* is $\mathsf{domain}(f) = A$ and *codomain* is $\mathsf{codomain}(f) = B$. The set of all functions from A to B is denoted by B^A or $[A \to B]$. The *image* of f is $\mathsf{image}(f) = \{f(a) \mid a \in A\}$. We call f an *injective function* or *injection* iff $f(a_1) = f(a_2)$ implies $a_1 = a_2$ for any $a_1, a_2 \in A$. We call f a *surjective function* or *surjection* iff $\mathsf{image}(f) = \mathsf{codomain}(f)$. We call f a *bijective function* or *bijection* iff it is both injective and surjective. For a subset $A_0 \subseteq A$, the *restriction* of f over A_0, denoted by $f|_{A_0} \colon A_0 \to B$, is a function defined by

$$f|_{A_0}(a) = f(a) \quad \text{for all } a \in A_0$$

A *partial function* from A to B is denoted by $f \colon A \rightharpoonup B$, where $\mathsf{domain}(f) \subseteq A$. Total functions are special instances of partial functions with $\mathsf{domain}(f) = A$. The set of all partial functions from A to B is denoted by $[A \rightharpoonup B]$. We write $f \colon A \rightharpoonup_{\mathsf{fin}} B$ to mean that $\mathsf{domain}(f)$ is finite. The set of all finite-domain partial functions from A to B is denoted by $[A \rightharpoonup_{\mathsf{fin}} B]$. We use $\emptyset$ to denote a partial function with an empty domain. For $a \in A \setminus \mathsf{domain}(f)$, we say that f is *undefined* at a, written $f(a) = \bot$.

For a function (or partial function) f from A to B, we use $f[b_0/a_0]$ where $a_0 \in A$ and

$b_0 \in B$ to denote the *updated function* (or *updated partial function*) f' such that $f'(a_0) = b_0$ and $f'(a) = f(a)$ for all $a \in A \setminus \{a_0\}$. For two functions (or partial functions) f and g, we write $f \overset{a_0}{\sim} g$ where $a_0 \in A$ to mean that $f(a) = g(a)$ for all $a \in A \setminus \{a_0\}$. We write $f \overset{A_0}{\sim} g$ where $A_0 \subseteq A$ to mean that $f \overset{a_0}{\sim} g$ for all $a \in A \setminus A_0$.

We say that partial functions $f, g\colon A \rightharpoonup B$ are *disjoint*, if $\mathsf{domain}(f) \cap \mathsf{domain}(g) = \emptyset$. For disjoint partial functions $f, g\colon A \rightharpoonup B$, their *disjoint union* is a partial function $f \mathbin{\dot\cup} g\colon A \rightharpoonup B$, given by

$$(f \mathbin{\dot\cup} g)(a) = \begin{cases} f(a) & \text{if } a \in \mathsf{domain}(f) \\ g(a) & \text{if } a \in \mathsf{domain}(g) \\ \bot & \text{otherwise} \end{cases}$$

Note that $\mathsf{domain}(f \mathbin{\dot\cup} g) = \mathsf{domain}(f) \mathbin{\dot\cup} \mathsf{domain}(g)$. For simplicity, we automatically require that f and g are disjoint whenever we write $f \mathbin{\dot\cup} g$.

Given $f\colon \mathcal{P}(A) \to \mathcal{P}(A)$, a fixed point or *fixpoint* of f is a set $A_0 \subseteq A$ such that $f(A_0) = A_0$. A *pre-fixpoint* (or *post-fixpoint*) of f is a set $A_0 \subseteq A$ such that $f(A_0) \subseteq A_0$ (or $A_0 \subseteq f(A_0)$). Thus, A_0 is a fixpoint iff it is a pre-fixpoint and a post-fixpoint. We say that f is *monotone* iff $A_1 \subseteq A_2$ implies $f(A_1) \subseteq f(A_2)$ for all $A_1, A_2 \subseteq A$.

Theorem 2.1 (Knaster-Tarski fixpoint theorem [63])**.** Let $f\colon \mathcal{P}(A) \to \mathcal{P}(A)$ be a monotone function. Then f has a unique least fixpoint $\mathbf{lfp}\, f$ and a unique greatest fixpoint $\mathbf{gfp}\, f$, given as follows:

$$\mathbf{lfp}\, f = \bigcap \{A_0 \subseteq A \mid f(A_0) \subseteq A_0\} \qquad \mathbf{gfp}\, f = \bigcup \{A_0 \subseteq A \mid A_0 \subseteq f(A_0)\}$$

In other words, the least fixpoint is also the least pre-fixpoint, and the great fixpoint is also the greatest post-fixpoint.

Let Λ be a set whose elements are called indices. A Λ-*indexed set* is denoted by $A = \{A_\lambda\}_{\lambda \in \Lambda}$, where A_λ is a set for each $\lambda \in \Lambda$. For simplicity, we often write $a \in A$ to mean that $a \in A_\lambda$ for some $\lambda \in \Lambda$. For two Λ-indexed sets $A = \{A_\lambda\}_{\lambda \in \Lambda}$ and $B = \{B_\lambda\}_{\lambda \in \Lambda}$, we write $f\colon A \to B$ to denote a Λ-*indexed function*, where $f(a) \in B_\lambda$ for every $\lambda \in \Lambda$ and $a \in A_\lambda$.

Definition 2.1. Given a function $f\colon A \to \mathcal{P}(B)$, we define its *pointwise extension* $f^{\text{ext}}\colon \mathcal{P}(A) \to \mathcal{P}(B)$ by

$$f^{\text{ext}}(A_0) = \bigcup_{a \in A_0} f(a) \qquad \text{for all } A_0 \subseteq A$$

Note that $f^{\text{ext}}(\emptyset) = \emptyset$. For any Λ-indexed set $\{A_\lambda\}_{\lambda \in \Lambda}$, $f^{\text{ext}}(\bigcup_{\lambda \in \Lambda} A_\lambda) = \bigcup_{\lambda \in \Lambda} f^{\text{ext}}(A_\lambda)$. In particular, we have $f^{\text{ext}}(A_1 \cup \cdots \cup A_n) = f^{\text{ext}}(A_1) \cup \cdots \cup f^{\text{ext}}(A_n)$ for any $A_1, \ldots, A_n \subseteq A$.

Given sets $A_1, \dots, A_n$ for $n \geq 1$, the set $A_1 \times \cdots \times A_n$ is the set of *n-tuples*, given by $A_1 \times \cdots \times A_n = \{(a_1, \dots, a_n) \mid a_i \in A_i, 1 \leq i \leq n\}$. Note that the 1-tuples are simply the elements in A_1. The 2-tuples are called *pairs*. An *n-ary relation* over $A_1, \dots, A_n$ is a subset of $A_1 \times \cdots \times A_n$. When $n = 1$, we call it a *unary relation*. When $n = 2$, we call it a *binary relation*. For an n-ary relation $R \subseteq A_1 \times \cdots \times A_n$, we say that $R(a_1, \dots, a_n)$ *holds* iff $(a_1, \dots, a_n) \in R$. The set of all n-ary relations over $A_1 \times \cdots \times A_n$ is $\mathcal{P}(A_1 \times \cdots \times A_n)$.

We define

$$A^n = \underbrace{A \times \cdots \times A}_{n} \qquad \text{for } n \geq 2$$

The set of all n-ary relations over A^n is $\mathcal{P}(A^n)$. The set of all relations over the elements/tuples of A is $\bigcup_{i \geq 1} \mathcal{P}(A^i)$.

The *identity relation* over A is denoted by id_A and defined by $\mathsf{id}_A = \{(a, a) \mid a \in A\}$. For relations $R_1 \subseteq A \times B$ and $R_2 \subseteq B \times C$, their *composition* is a relation $R_1 \circ R_2 \subseteq A \times C$, defined by

$$R_1 \circ R_2 = \{(a, c) \in A \times C \mid \text{there exists } b \in B \text{ such that } R_1(a, b) \text{ holds and } R_2(b, c) \text{ holds}\}$$

Note that $\circ$ is associative so we can write $R_1 \circ \cdots \circ R_n$ without needing to put parentheses.

For a relation $R \subseteq A \times A$, we say that R is

1. *reflexive*, if $R(a, a)$ holds for all $a \in A$;
2. *symmetric*, if $R(a_1, a_2)$ holds implies $R(a_2, a_1)$ holds for all $a_1, a_2 \in A$;
3. *transitive*, if $R(a_1, a_2)$ holds and $R(a_2, a_3)$ holds implies $R(a_1, a_3)$ holds for all $a_1, a_2, a_3 \in A$.

We let $R^n = \underbrace{R \circ \cdots \circ R}_{n}$. When $n = 1$, we let $R^1 = R$. When $n = 0$, we let $R^0 = \mathsf{id}_A$. The *transitive closure* of R is denoted by R^+ and defined by $R^+ = \bigcup_{i \geq 1} R^i$. The *reflexive and transitive closure* of R is denoted by R^* and defined by $R^* = \bigcup_{i \geq 0} R^i$.

2.2 FIRST-ORDER LOGIC

We review (many-sorted) first-order logic, abbreviated as FOL.

Definition 2.2. A *FOL signature* (S, F, Π) consists of a set S of *sorts*, an $(S^* \times S)$-indexed set $F = \{F_{s_1 \dots s_n, s}\}_{s_1, \dots, s_n, s \in S}$ of *function symbols*, and an S^+-indexed set $\Pi = \{\Pi_{s_1 \dots s_n}\}_{s_1, \dots, s_n \in S}$ of *predicate symbols*.

Definition 2.3. Given a FOL signature (S, F, Π) and an S-indexed set $V = \{V_s\}_{s \in S}$ of *(many-sorted) variables*, denoted by $x : s$, $y : s$, etc, the syntax of FOL is given by the following grammar:

$$\begin{array}{rrl}
\underline{\text{FOL terms}} & t_s ::= & x : s \in V_s \\
 & & \mid f(t_{s_1}, \ldots, t_{s_n}) \quad \text{with } f \in F_{s_1 \ldots s_n, s} \\
\underline{\text{FOL formulas}} & \varphi ::= & \pi(t_{s_1}, \ldots, \pi_{s_n}) \quad \text{with } \pi \in \Pi_{s_1 \ldots s_n} \\
 & & \mid \varphi_1 \wedge \varphi_2 \\
 & & \mid \neg \varphi \\
 & & \mid \exists x : s \,.\, \varphi
\end{array}$$

We use $\mathit{freeVar}(\varphi)$ to denote the set of free variables in φ. We write $\varphi[t_s/x : s]$ for the result of substituting t_s for $x : s$ in φ, where α-renaming happens implicitly to avoid variable capture.

Definition 2.4. Given a FOL signature (S, F, Π), a *FOL* (S, F, Π)*-model* or simply *FOL model* is a tuple

$$M = (\{M_s\}_{s \in S}, \{f_M\}_{f \in F}, \{\pi_M\}_{\pi \in \Pi})$$

where

1. M_s is a carrier set for every $s \in S$;

2. $f_M \colon M_{s_1} \times \cdots \times M_{s_n} \to M_s$ is a function for every $f \in F_{s_1 \ldots s_n, s}$;

3. $\pi_M \subseteq M_{s_1} \times \cdots \times M_{s_n}$ is a relation for every $\pi \in \Pi_{s_1 \ldots s_n}$.

Definition 2.5. Let M be a FOL model. A *FOL* M*-valuation* or simply *FOL valuation* $\rho \colon V \to M$ is a function such that $\rho(x : s) \in M_s$ for every $s \in S$ and $x : s \in V_s$. The *term extension* of ρ is a function $\bar{\rho}$ from the set of FOL terms to M, defined by

1. $\bar{\rho}(x : s) = \rho(x : s)$;

2. $\bar{\rho}(f(t_{s_1}, \ldots, t_{s_n})) = f_M(\bar{\rho}(t_{s_1}), \ldots, \bar{\rho}(t_{s_n}))$.

Note that $\bar{\rho}(t_s) \in M_s$ for every FOL term t_s whose sort is s.

Definition 2.6. Let M be a FOL model. The *FOL satisfaction relation* $M, \rho \vDash_{\mathsf{FOL}} \varphi$ is defined for all ρ as follows:

1. $M, \rho \vDash_{\mathsf{FOL}} \pi(t_{s_1}, \ldots, t_{s_n})$ iff $\pi_M(\bar{\rho}(t_{s_1}), \ldots, \bar{\rho}(t_{s_n}))$ holds;

(Propositional Tautology)	φ, if φ is a propositional tautology
(Modus Ponens)	$\dfrac{\varphi_1 \quad \varphi_1 \to \varphi_2}{\varphi_2}$
(Term Substitution)	$\varphi[t_s/x:s] \to \exists x:s\,.\,\varphi$
($\exists$-Generalization)	$\dfrac{\varphi_1 \to \varphi_2}{(\exists x:s\,.\,\varphi_1) \to \varphi_2}$ if $x:s \notin \mathit{freeVar}(\varphi_2)$

Figure 2.1: Sound and Complete Proof System of FOL

2. $M, \rho \vDash_{\mathsf{FOL}} \varphi_1 \land \varphi_2$ iff $M, \rho \vDash_{\mathsf{FOL}} \varphi_1$ and $M, \rho \vDash_{\mathsf{FOL}} \varphi_2$;

3. $M, \rho \vDash_{\mathsf{FOL}} \neg\varphi$ iff $M, \rho \nvDash_{\mathsf{FOL}} \varphi$;

4. $M, \rho \vDash_{\mathsf{FOL}} \exists x:s\,.\,\varphi$ iff there exists $a \in M_s$ such that $M, \rho[a/x:s] \vDash_{\mathsf{FOL}} \varphi$.

We write $M \vDash_{\mathsf{FOL}} \varphi$ iff $M, \rho \vDash_{\mathsf{FOL}} \varphi$ for all ρ. A *FOL theory* Γ is a set of FOL formulas/axioms. We write $M \vDash_{\mathsf{FOL}} \Gamma$ iff $M \vDash_{\mathsf{FOL}} \psi$ for all $\psi \in \Gamma$. We write $\Gamma \vDash_{\mathsf{FOL}} \varphi$ iff $M \vDash_{\mathsf{FOL}} \Gamma$ implies $M \vDash_{\mathsf{FOL}} \varphi$ for all M.

FOL has a sound and complete Hilbert-style proof system as shown in Figure 2.1. The corresponding provability relation is denoted by $\Gamma \vdash_{\mathsf{FOL}} \varphi$.

Theorem 2.2. For any FOL theory Γ and formula φ, $\Gamma \vDash_{\mathsf{FOL}} \varphi$ iff $\Gamma \vdash_{\mathsf{FOL}} \varphi$.

2.3 FIRST-ORDER LOGIC WITH LEAST FIXPOINTS

We review first-order logic with least fixpoints, abbreviated as LFP. LFP is an extension of FOL with predicate variables and an operator lfp that builds least fixpoints.

Definition 2.7. An *LFP signature* (S, F, Π) is the same as a FOL signature.

Definition 2.8. Given an LFP signature (S, F, Π), an S-indexed set $EV = \{EV_s\}_{s \in S}$ of *element variables*, and an S^+-indexed set $PV = \{PV_{s_1 \ldots s_n}\}_{s_1,\ldots,s_n \in S}$ of *predicate variables*, the syntax of LFP extends the syntax of FOL with the following grammar rules:

$$
\begin{array}{lll}
\underline{\text{LFP formulas}} \quad \varphi ::= \text{(syntax of FOL formulas)} & & \\
\qquad\qquad\qquad \mid P(t_{s_1}, \ldots, t_{s_n}) & & \text{with } P \in PV_{s_1 \ldots, s_n} \\
\qquad\qquad\qquad \mid [\mathsf{lfp}_{P, x_1 : s_1 \ldots, x_n : s_n} \varphi](t_{s_1}, \ldots, t_{s_n}) & & \text{with } P \in PV_{s_1 \ldots, s_n}
\end{array}
$$

where $[\mathsf{lfp}_{P,x_1\,:\,s_1\ldots,x_n\,:\,s_n}\varphi](t_{s_1},\ldots,t_{s_n})$ requires that φ is positive in P, that is, every subformula of φ that has form $P(t'_{s_1},\ldots,t'_{s_n})$ must occur under an even number of $\neg$'s.

Definition 2.9. An LFP model M is the same as a FOL model. An *LFP M-valuation* or simply an *LFP valuation* $\rho = (\rho_{EV}, \rho_{PV})$ is a pair, where $\rho_{EV}\colon V \to M$ and $\rho_{PV}\colon PV \to \{\mathcal{P}(M_{s_1}\times\cdots\times M_{s_n})\}_{s_1,\ldots,s_n\in S}$. That is, $\rho_{PV}(P)$ is an n-ary relation over $M_{s_1},\ldots,M_{s_n}$, for $P \in PV_{s_1\ldots s_n}$. Let $\overline{\rho_{EV}}$ be the term extension of ρ_{EV} in Definition 2.5. The *LFP satisfaction relation* $M,\rho \vDash_{\mathsf{LFP}} \varphi$ is defined for all ρ by extending the FOL satisfaction relation with two additional rules:

1. $M,\rho \vDash_{\mathsf{LFP}} P(t_{s_1},\ldots,t_{s_n})$ iff $\rho_{PV}(P)(\overline{\rho_{EV}}(t_{s_1}),\ldots,\overline{\rho_{EV}}(t_{s_n}))$ holds;
2. $M,\rho \vDash_{\mathsf{LFP}} [\mathsf{lfp}_{P,x_1\,:\,s_1\ldots,x_n\,:\,s_n}\varphi](t_{s_1},\ldots,t_{s_n})$ iff $(\overline{\rho_{EV}}(t_{s_1}),\ldots,\overline{\rho_{EV}}(t_{s_n})) \in \bigcap\{R \subseteq M_{s_1}\times\cdots\times M_{s_n} \mid$ for all $a_i \in M_{s_i}$, $1 \le i \le n$, $M,\rho[R/P, a_1/x_1 : s_1,\ldots,a_n/x_n : s_n] \vDash_{\mathsf{LFP}} \varphi$ implies $(a_1,\ldots,a_n) \in R\}$.

We write $M \vDash_{\mathsf{LFP}} \varphi$ iff $M,\rho \vDash_{\mathsf{LFP}} \varphi$ for all ρ and all the predicate variables of φ are bound by lfp. We write $\vDash_{\mathsf{LFP}} \varphi$ iff $M \vDash_{\mathsf{LFP}} \varphi$ for all M.

Our presentation of LFP is slightly different from the classical presentation. The classical presentation enforces all the predicate variables in an LFP formula to be bound by lfp. The semantics of predicate variables, which are needed for defining the semantics of $[\mathsf{lfp}_{R,x_1\,:\,s_1\ldots,x_n\,:\,s_n}\varphi]$, are given by a model of an extended signature, where all the predicate variables are added as predicate symbols and interpreted as relations. In our presentation, we do not extend the signature nor the model. Instead, we extend the valuation with ρ_{PV}, which maps predicate variables to relations. This modified yet equivalent presentation of LFP fits better in Section 5.2, where we will define LFP in matching μ-logic.

2.4 SECOND-ORDER LOGIC

We review (many-sorted) second-order logic, abbreviated as SOL.

Definition 2.10. A *SOL signature* (S, C, Π) consists of a set S of sorts, an S-indexed set C of *constant symbols* that are function symbols of arity 0, and an S^*-indexed set Π of predicate symbols.

Definition 2.11. Given a SOL signature (S, C, Π), an S-indexed set $EV = \{EV_s\}_{s\in S}$ of element variables, and an S^+-indexed set $PV = \{PV_{s_1\ldots s_n}\}_{s_1,\ldots,s_n\in S}$ of predicate variables, the syntax of SOL is given by the following grammar:

$$\underline{\text{SOL terms}}\quad t_s ::= x : s \in EV_s$$

$$\begin{array}{lrl} & & \mid c \in C_s \\ \underline{\text{SOL formulas}} & \varphi ::= & t_s = t'_s \\ & & \mid \pi(t_{s_1}, \ldots, t_{s_n}) \quad \text{with } \pi \in \Pi_{s_1 \ldots s_n} \\ & & \mid P(t_{s_1}, \ldots, t_{s_n}) \quad \text{with } P \in PV_{s_1 \ldots s_n} \\ & & \mid \varphi_1 \wedge \varphi_2 \\ & & \mid \neg \varphi \\ & & \mid \exists x : s \,.\, \varphi \\ & & \mid \exists P \,.\, \varphi \quad \text{with } P \in PV_{s_1 \ldots s_n} \end{array}$$

Definition 2.12. Given a SOL signature (S, C, Π), a *SOL* (S, C, Π)*-model* or simply *SOL model* $M = (\{M_s\}_{s \in S}, \{c_M\}_{c \in C}, \{\pi_M\}_{\pi \in \Pi})$ is a tuple, where

1. M_s is a carrier set for every $s \in S$;
2. $c_M \in M_s$ for every $c \in C_s$;
3. $\pi_M \subseteq M_{s_1} \times \cdots \times M_{s_n}$ for every $\pi \in \Pi_{s_1 \ldots s_n}$.

Definition 2.13. Given a SOL model M, a *SOL* M*-valuation* or simply *SOL valuation* $\rho = (\rho_{EV}, \rho_{PV})$ is the same as Definition 2.9. The SOL satisfaction relation $M, \rho \vDash_{\mathsf{SOL}} \varphi$ is defined for all ρ by the following rules:

1. $M, \rho \vDash_{\mathsf{SOL}} t_s = t'_s$ iff $\overline{\rho_{EV}}(t_s) = \overline{\rho_{EV}}(t'_s)$;
2. $M, \rho \vDash_{\mathsf{SOL}} \pi(t_{s_1}, \ldots, t_{s_n})$ iff $\pi_M(\overline{\rho_{EV}}(t_{s_1}), \ldots, \overline{\rho_{EV}}(t_{s_n}))$ holds;
3. $M, \rho \vDash_{\mathsf{SOL}} P(t_{s_1}, \ldots, t_{s_n})$ iff $\rho_{PV}(P)(\overline{\rho_{EV}}(t_{s_1}), \ldots, \overline{\rho_{EV}}(t_{s_n}))$ holds;
4. $M, \rho \vDash_{\mathsf{SOL}} \varphi_1 \wedge \varphi_2$ iff $M, \rho \vDash_{\mathsf{SOL}} \varphi_1$ and $M, \rho \vDash_{\mathsf{SOL}} \varphi_2$;
5. $M, \rho \vDash_{\mathsf{SOL}} \neg \varphi$ iff $M, \rho \nvDash_{\mathsf{SOL}} \varphi$;
6. $M, \rho \vDash_{\mathsf{SOL}} \exists x : s \,.\, \varphi$ iff there exists $a \in M_s$ such that $M, \rho[a/x : s] \vDash_{\mathsf{SOL}} \varphi$;
7. $M, \rho \vDash_{\mathsf{SOL}} \exists P \,.\, \varphi$ iff there exists $R \subseteq M_{s_1} \times \cdots \times M_{s_n}$ such that $M, \rho[R/P] \vDash_{\mathsf{SOL}} \varphi$, where $P \in PV_{s_1 \ldots s_n}$.

We write $M \vDash_{\mathsf{SOL}} \varphi$ iff $M, \rho \vDash_{\mathsf{SOL}} \varphi$ for all ρ. A *SOL theory* Γ is a set of SOL formulas/axioms. We write $M \vDash_{\mathsf{SOL}} \Gamma$ iff $M \vDash_{\mathsf{SOL}} \psi$ for all $\psi \in \Gamma$. We write $\Gamma \vDash_{\mathsf{SOL}} \varphi$ iff $M \vDash_{\mathsf{SOL}} \Gamma$ implies $M \vDash_{\mathsf{SOL}} \varphi$ for all M.

Monadic SOL, abbreviated as MSO, is an instance of SOL where all predicate variables are unary, i.e., taking only one argument.

2.5 EQUATIONAL SPECIFICATIONS AND INITIAL ALGEBRA SEMANTICS

Equational specifications (also known as algebraic specifications) and initial algebra semantics provide a generic and principled framework to study induction. We review the main definitions and notation about them following the standard many-sorted approach [19, 20].

Definition 2.14. Given a many-sorted signature (S, F), an (S, F)-*algebra* or simply F-*algebra* $A = (\{A_s\}_{s\in S}, \{f_A\}_{f\in F})$ is a pair, where

1. A_s is a carrier set for every $s \in S$;

2. $f_A \colon : A_{s_1} \times \cdots \times A_{s_n} \to A_s$ is a function for every $f \in F_{s_1\ldots s_n,s}$.

Given an S-indexed set $V = \{V_s\}_{s\in S}$ of variables, the syntax of *(S,F)-terms* or simply F-*terms* is the same as Definition 2.3. Let $T_F(V) = \{T_{F,s}(V)\}_{s\in S}$ be an S-indexed set of terms. The set $T_F(\emptyset) = \{T_{F,s}(\emptyset)\}_{s\in S}$ includes all the *ground terms*, i.e., terms with no variables. We often abbreviate $T_F(\emptyset)$ as T_F and $T_{F,s}(\emptyset)$ as $T_{F,s}$. An A-*valuation* $\rho \colon V \to A$ and its term extension $\bar{\rho}$ are the same as Definition 2.5. When $V = \emptyset$, there is a unique trivial valuation $\emptyset \colon \emptyset \to A$. We define $\mathsf{eval}_A(t) = \bar{\emptyset}(t)$ for $t \in T_F$ and feel free to drop the subscript A when it is understood or not important.

Definition 2.15. Given a many-sorted signature (S, F), an (S, F)-*equation* or simply F-*equation* is written $\forall V \,.\, t_s = t'_s$, where V is finite and $t_s, t'_s \in T_{F,s}(V)$. A *ground* (S, F)-*equation* or simply *ground* F-*equation* $\forall \emptyset \,.\, t_s = t'_s$ is when $V = \emptyset$ and $t_s, t'_s \in T_{F,s}$.

Definition 2.16. Given an (S, F)-algebra A and an (S, F)-equation $\forall V \,.\, t = t'$, we write $A \vDash_{\mathsf{EQ}} \forall V \,.\, t = t'$ iff $\bar{\rho}(t) = \bar{\rho}(t')$ for all ρ. An *equational specification* (S, F, E) consists of a many-sorted signature (S, F) and a set E of (S, F)-equations. We often abbreviate (S, F, E) as (F, E) or simply E. We write $A \vDash_{\mathsf{EQ}} E$ iff $A \vDash_{\mathsf{EQ}} e$ for all $e \in E$, and we call A an (S, F, E)-*algebra* or simply (F, E)-*algebra* or E-*algebra*. We write $E \vDash_{\mathsf{EQ}} e$ iff $A \vDash_{\mathsf{EQ}} e$ for all E-algebras A.

Next, we review rules of equational deduction for many-sorted algebras. There are many equivalent definitions of equational deduction in the literature. We present one standard definition in Figure 2.2 and denote the corresponding provability relation by $E \vdash_{\mathsf{EQ}} e$.

Theorem 2.3. For any equational specification E and equation e, $E \vdash_{\mathsf{EQ}} e$ iff $E \vDash_{\mathsf{EQ}} e$.

Definition 2.17. Let A be an (S, F)-algebra. A *congruence* on A is an S-indexed set $R = \{R_s\}_{s\in S}$ of equivalence relations $R_s \subseteq A_s \times A_s$ for $s \in S$, such that $R_{s_i}(a_i, b_i)$ holds for all $1 \leq i \leq n$ implies $R_s\,(f_A(a_1, \ldots, a_n), f_A(b_1, \ldots, b_n))$ holds, for all $f \in F_{s_1\ldots s_n,s}$ and $a_i, b_i \in A_{s_i}$,

$$\begin{array}{ll}
(\textsc{Axiom}) & \forall V\,.\,t_s = t'_s \quad \text{if } (\forall V\,.\,t_s = t'_s) \in E \\[2ex]
(\textsc{Reflexivity}) & \forall V\,.\,t_s = t_s \\[1ex]
(\textsc{Symmetry}) & \dfrac{\forall V\,.\,t_s = t'_s}{\forall V\,.\,t'_s = t_s} \\[2ex]
(\textsc{Transitivity}) & \dfrac{\forall V\,.\,t_s = t'_s \quad \forall V\,.\,t'_s = t''_s}{\forall V\,.\,t_s = t''_s} \\[2ex]
(\textsc{Congruence}) & \dfrac{\forall V\,.\,t_{s_1} = t'_{s_1} \quad \ldots \quad \forall V\,.\,t_{s_n} = t'_{s_n}}{\forall V\,.\,f(t_{s_1}, \ldots, t_{s_n}) = f(t'_{s_1}, \ldots, t'_{s_n})} \\[2ex]
(\textsc{Substitution}) & \dfrac{\forall V\,.\,t_s = t'_s}{\forall U\,.\,t_s\theta = t'_s\theta} \text{with substitution } \theta \colon V \to T_F(U)
\end{array}$$

Figure 2.2: Sound and Complete Proof Rules for Equational Deduction

$1 \leq i \leq n$. The *R-quotient algebra of A* is an (S, F)-algebra $A_{/R} = (\{A_{/R,s}\}_{s \in S}, \{f_{A_{/R}}\}_{f \in F})$, where

1. $A_{/R,s} = \{[a]_R \mid a \in A_s\}$ for every $s \in S$; here, $[a]_R = \{b \in A_s \mid R_s(a, b) \text{ holds}\}$ is the R-equivalence class of a;

2. $f_{A_{/R}} \colon A_{/R,s_1} \times \cdots \times A_{/R,s_n} \to A_{/R,s}$ is defined by $f_{A_{/R}}([a_1]_R, \ldots, [a_n]_R) = [f_A(a_1, \ldots, a_n)]_R$ for all $[a_i]_R \in A_{/R,s_i}$, $1 \leq i \leq n$, for every $f \in F_{s_1 \ldots s_n, s}$.

Note that $f_{A_{/R}}$ is well-defined because R is a congruence.

Definition 2.18. Given a many-sorted signature (S, F), an (S, F)-*term algebra* or simply F-*term algebra* $T_F = (\{T_{F,s}\}_{s \in S}, \{f_{T_F}\}_{f \in F})$ is an F-algebra, where

1. $T_{F,s}$ is the set of ground terms of sort s, for every $s \in S$;

2. $f_{T_F} \colon T_{F,s_1} \times \cdots \times T_{F,s_n} \to T_{F,s}$ is defined by $f_{T_F}(t_{s_1}, \ldots, t_{s_n}) = f(t_{s_1}, \ldots, t_{s_n})$ for all $t_{s_i} \in T_{F,s_i}$, $1 \leq i \leq n$, for every $f \in F_{s_1 \ldots s_n, s}$.

Equational deduction generates a congruence on T_F.

Proposition 2.4. *Let E be an equational specification. Define a relation $\simeq_{E,s} \subseteq T_{F,s} \times T_{F,s}$ such that $t_s \simeq_{E,s} t'_s$ iff $E \vdash_{\mathsf{EQ}} \forall \emptyset\,.\,t_s = t'_s$, for all $t_s, t'_s \in T_{F,s}$. Then, $\simeq_E = \{\simeq_{E,s}\}_{s \in S}$ is a congruence on T_F.*

We use $[t_s]_{\simeq_E}$, or simply $[t_s]_E$ or $[t_s]$, to denote the set of terms that are provably equal to t. We abbreviate $t_s \simeq_{E,s} t'_s$ as $t_s \simeq_E t'_s$ or $t_s \simeq t'_s$.

Definition 2.19. Given an equational specification (S, F, E), the (S, F, E)-*quotient term algebra* or simply (F, E)-*quotient term algebra* or E-*quotient term algebra*, written $T_{F/E}$, is the $\simeq_E$-quotient algebra of T_F. Specifically, $T_{F/E} = (\{T_{F/E,s}\}_{s\in S}, \{f_{T_{F/E}}\}_{f\in F})$, where

1. $T_{F/E,s} = \{[t_s]_E \mid t_s \in T_{F,s}\}$ for every $s \in S$;

2. $f_{T_{F/E}} \colon T_{F/E,s_1}\times\cdots\times T_{F/E,s_n} \to T_{F/E,s}$ is defined by $f_{T_{F/E}}([t_{s_1}], \ldots, [t_{s_n}]) = [f(t_{s_1}, \ldots, t_{s_n})]$ for all $[t_{s_i}] \in T_{F/E,s_i}$, $1 \leq i \leq n$, for every $f \in F_{s_1\ldots s_n,s}$.

Term algebras and quotient term algebras are the concrete examples of initial algebras, which are initial objects in the category of algebras. We first review the definition of algebra morphisms.

Definition 2.20. For (S, F)-algebras A and B, an *(algebra) morphism* is a function $h\colon A \to B$ such that $h(f_A(a_1, \ldots, a_n)) = f_B(h(a_1), \ldots, h(a_n))$ for all $f \in F_{s_1\ldots s_n,s}$ and $a_i \in A_{s_i}$, $1 \leq i \leq n$. If h is a morphism and its inverse $h^{-1}\colon B \to A$ exists, then h is an *isomorphism* and A and B are *isomorphic.*

Definition 2.21. Given an equational specification (S, F, E), an *initial* (S, F, E)-*algebra* or simply *initial* (F, E)-*algebra* or *initial* E-*algebra*, is an (S, F)-algebra I such that for every (F, E)-algebra A, there exists a unique morphism $h_A\colon I \to A$. An initial $(S, F, \emptyset)$-algebra is called an *initial* (S, F)-*algebra* or simply *initial* F-*algebra.*

Theorem 2.5. Any two initial (F, E)-algebras are isomorphic. In particular, T_F is an initial F-algebra and $T_{F/E}$ is an initial (F, E)-algebra.

An equational specification states the existence of some data, operations, and equational properties. Its initial algebras are the minimal realization of the specification, in the sense that all of its elements are representable by terms and all the (equational) properties are derivable from E. An equivalent characterization of initiality is the famous “no junk, no confusion” slogan, firstly proposed in [21].

Theorem 2.6. Let A be an (S, F, E)-algebra. We define no-confusion and no-junk as follows:

1. A satisfies *no-confusion*, iff $A \vdash_{\mathsf{EQ}} \forall\emptyset\,.\,t_s = t'_s$ implies $E \vdash_{\mathsf{EQ}} \forall\emptyset\,.\,t_s = t'_s$ for all $t_s, t'_s \in T_{F,s}$.

2. A satisfies *no-junk*, iff for any $a \in A_s$ there exists $t_a \in T_{F,s}$ such that $\mathsf{eval}_A(t_a) = a$.

Then, A is an initial (S, F, E)-algebra iff it satisfies no-junk and no-confusion.

If A satisfies no-confusion, then for any $t_s, t'_s \in T_{F,s}$ that have the same semantics in A, we have $E \vdash_{\mathsf{EQ}} \forall\emptyset \,.\, t_s = t'_s$, which implies that $E \vDash_{\mathsf{EQ}} \forall\emptyset \,.\, t_s = t'_s$ (Theorem 2.3). In other words, if t_s and t'_s have the same semantics in A, then they have the same semantics in all E-algebras. On the other hand, if A satisfies no-junk, then every element in A is representable by a ground term.

2.6 SEPARATION LOGIC

Separation logic [22], abbreviated as SL, is a logic specifically crafted for reasoning about heap structures. SL has many variants; the formalization that we consider here is adapted from [23]. The most characteristic construct in SL is separating conjunction $\varphi_1 * \varphi_2$, which specifies a conjunctive heap of two disjoint heaps. In addition, SL has the model of heaps (i.e., finite-domain maps) hard-wired in its semantics, which makes it a logic specifically crafted for heap reasoning.

Definition 2.22. Let V be a set of variables and *RSymb* be a finite set of *recursive symbols*. For each $P \in \mathit{RSymb}$ we use $\mathit{arity}(P) \geq 1$ to denote its arity. The syntax of SL is given by the following grammar:

$$\begin{aligned}
&\underline{\text{SL terms}} \quad t ::= x \in V \\
&\qquad\qquad\qquad \mid \mathsf{nil} \\
&\underline{\text{SL formulas:}} \quad \varphi ::= \text{(syntax of FOL formulas)} \\
&\qquad\qquad\qquad \mid \mathsf{emp} && \text{// the empty heap} \\
&\qquad\qquad\qquad \mid t_1 \mapsto t_2 && \text{// singleton heaps} \\
&\qquad\qquad\qquad \mid \varphi_1 * \varphi_2 && \text{// separating conjunction} \\
&\qquad\qquad\qquad \mid \varphi_1 \mathrel{-\!\!*} \varphi_2 && \text{// separating implication (the ``magic wand'')} \\
&\qquad\qquad\qquad \mid P(t_1, \dots, t_n) \text{ with } P \in \mathit{RSymb} \text{ and } \mathit{arity}(p) = n
\end{aligned}$$

A *recursive symbol definition* D is a set as follows:

$$D = \{P(x_1, \dots, x_n) =_{\mathbf{lfp}} \psi_P \mid P \in \mathit{RSymb} \text{ and } \mathit{arity}(P) = n\}$$

where $\mathit{freeVar}(\psi_P) \subseteq \{x_1, \dots, x_n\}$ and ψ_P is positive in P, the same as LFP (Definition 2.8). In this work, we do not consider mutually recursive symbols, so we require that P is the only recursive predicate symbol in ψ_P.

Many heap structures, especially those featuring induction, can be defined using recursive symbol definitions. For example, singly-linked lists can be defined by a recursive symbol list and the following recursive symbol definition:

$$\mathsf{list}(x) =_{\mathbf{lfp}} (x = \mathsf{nil}) \land \mathsf{emp} \lor \exists y \,.\, (x \neq \mathsf{nil}) \land x \mapsto y * \mathsf{list}(y)$$

Intuitively, if $x = \mathsf{nil}$ then $\mathsf{list}(\mathsf{nil})$ specifies the empty heap emp. Otherwise, there exists y such that x points to y (i.e., $x \mapsto y$), and in a separate heap segment there is a singly-linked list starting at y.

Definition 2.23. A *heap* is a partial function $h \colon \mathbb{N}^+ \rightharpoonup_{\text{fin}} \mathbb{N}$. The set of all heaps $\mathbb{H} = [\mathbb{N}^+ \rightharpoonup_{\text{fin}} \mathbb{N}]$. For heaps h_1 and h_2, their disjoint union is denoted by $h_1 \dot\cup h_2$ (Section 2.1). A *store* is a function $s \colon V \to \mathbb{N}$. Its term extension $\bar{s} \colon V \to \mathbb{N}$ is given by $\bar{s}(x) = s(x)$ for all $x \in V$ and $\bar{s}(\mathsf{nil}) = 0$.

Definition 2.24. Given a set of recursive symbols *RSymb* and a recursive symbol definition D in Definition 2.22, the SL satisfaction relation $s, h \vDash_{\mathsf{SL}} \varphi$ is defined for all heaps h and stores s by extending the FOL satisfaction relation with the following additional rules:

1. $s, h \vDash_{\mathsf{SL}} \mathsf{emp}$ iff $\mathsf{domain}(h) = \emptyset$;
2. $s, h \vDash_{\mathsf{SL}} t_1 \mapsto t_2$ iff $\bar{s}(t_1) \neq 0$, $\mathsf{domain}(h) = \{\bar{s}(t_1)\}$, and $h(\bar{s}(t_1)) = \bar{s}(t_2)$;
3. $s, h \vDash_{\mathsf{SL}} \varphi_1 * \varphi_2$ iff there exist h_1, h_2 such that $s, h_1 \vDash_{\mathsf{SL}} \varphi_1$, $s, h_2 \vDash_{\mathsf{SL}} \varphi_2$, and $h = h_1 \dot\cup h_2$;
4. $s, h \vDash_{\mathsf{SL}} \varphi_1 \mathrel{-\!\!*} \varphi_2$ iff for all h' such that h, h' are disjoint and $s, h' \vDash_{\mathsf{SL}} \varphi_1$, we have $s, h \dot\cup h' \vDash_{\mathsf{SL}} \varphi_2$;
5. $s, h \vDash_{\mathsf{SL}} P(t_1, \ldots, t_n)$ iff $(\bar{s}(t_1), \ldots, \bar{s}(t_n), h) \in |P|^{\mathsf{SL}}$, for $P \in \mathit{RSymb}$ and $\mathit{arity}(P) = n$,

where $|P|^{\mathsf{SL}}$ is given as follows. Define a function $\mathcal{F}_P \colon \mathcal{P}(\mathbb{N}^n \times \mathbb{H}) \to \mathcal{P}(\mathbb{N}^n \times \mathbb{H})$ by letting

$$\mathcal{F}_P(R) = \{(\bar{s}(x_1), \ldots, \bar{s}(x_n), h) \mid s, h \vDash_{\mathsf{SL},P} \psi_P\} \qquad \text{for } R \subseteq \mathbb{N}^n \times \mathbb{H}$$

where $\vDash_{\mathsf{SL},P}$ is the same as $\vDash_{\mathsf{SL}}$ except that $|P|^{\mathsf{SL}} = R$. Since ψ_P is positive in P, $\mathcal{F}_P$ is monotone and its unique least fixpoint $\mathbf{lfp}\,\mathcal{F}_p$ is given by Theorem 2.1. We let $|P|^{\mathsf{SL}} = \mathbf{lfp}\,\mathcal{F}_P$. Given a SL formula φ, we write $\vDash_{\mathsf{SL}} \varphi$ iff $s, h \vDash_{\mathsf{SL}} \varphi$ for all s and h.

2.7 MODAL LOGIC K

Modal logic is a big family of logics, with many variants and extensions. The formalization we consider here is called modal logic K with multiple modalities. We should not confuse modal logic K with the $\mathbb{K}$ formal semantics framework (Section 2.15). We will always use the blackboard bold letter $\mathbb{K}$ to refer to the latter.

Definition 2.25. Let AP be a set of *atomic propositions*, also known as propositional variables in the literature. Let L be a set of *labels*. The syntax of modal logic is given by the following grammar:

$$\begin{array}{ll} \underline{\text{modal logic K formulas}} & \varphi ::= p \in AP \\ & \quad\;\; \mid \varphi_1 \wedge \varphi_2 \\ & \quad\;\; \mid \neg\varphi \\ & \quad\;\; \mid [a]\varphi \text{ with } a \in L \end{array}$$

For each $a \in L$ the dual of $[a]$ is the modal operator $\langle a \rangle$, given by $\langle a \rangle\varphi \equiv \neg[a]\neg\varphi$.

Modal logic models are labeled transition systems.

Definition 2.26. Given a label set L, an *L-labeled transition system* or simply *labeled transition system* is a tuple $T = (S, \{\xrightarrow{a}\}_{a \in L})$, where S is a set of *states* and $\xrightarrow{a} \subseteq S \times S$ is a binary relation, called a *transition relation*, for every $a \in L$. We write $s_1 \xrightarrow{a} s_2$ to mean that $(\xrightarrow{a})(s_1, s_2)$ holds, for $s_1, s_2 \in S$.

Definition 2.27. Given a label set L and an L-labeled transition system $T = (S, \{\xrightarrow{a}\}_{a \in L})$, a *$T$-valuation* is a function $\rho \colon AP \to \mathcal{P}(S)$. The modal logic satisfaction relation $T, \rho, s \vDash_{\mathsf{K}} \varphi$ is defined for all ρ and $s \in S$ as follows:

1. $T, \rho, s \vDash_{\mathsf{K}} p$ iff $s \in \rho(p)$;
2. $T, \rho, s \vDash_{\mathsf{K}} \varphi_1 \wedge \varphi_2$ iff $T, \rho, s \vDash_{\mathsf{K}} \varphi_1$ and $T, \rho, s \vDash_{\mathsf{K}} \varphi_2$;
3. $T, \rho, s \vDash_{\mathsf{K}} \neg\varphi$ iff $T, \rho, s \nvDash_{\mathsf{K}} \varphi$;
4. $T, \rho, s \vDash_{\mathsf{K}} [a]\varphi$ iff for all $s' \in S$, $s \xrightarrow{a} s'$ implies $T, \rho, s' \vDash_{\mathsf{K}} \varphi$.

The derived semantics for $\langle a \rangle\varphi$ is

$$T, \rho, s \vDash_{\mathsf{K}} \langle a \rangle\varphi \text{ iff there exists } s' \in S \text{ such that } s \xrightarrow{a} s' \text{ and } T, \rho, s' \vDash_{\mathsf{K}} \varphi$$

(Propositional Tautology)	φ, if φ is a propositional tautology
(Modus Ponens)	$\dfrac{\varphi_1 \quad \varphi_1 \to \varphi_2}{\varphi_2}$
(K)	$[a](\varphi_1 \to \varphi_2) \to ([a]\varphi_1 \to [a]\varphi_2)$
(N)	$\dfrac{\varphi}{[a]\varphi}$

Figure 2.3: Sound and Complete Proof System of Modal Logic K

We write $T, \rho \vDash_{\mathsf{K}} \varphi$ iff $T, \rho, s \vDash_{\mathsf{K}} \varphi$ for all $s \in S$. We write $T \vDash_{\mathsf{K}} \varphi$ iff $T, \rho \vDash_{\mathsf{K}} \varphi$ for all ρ. We write $\vDash_{\mathsf{K}} \varphi$ iff $T \vDash_{\mathsf{K}} \varphi$ for all T.

Modal logic K has a sound and complete proof system as shown in Figure 2.3. In the literature, (K) is also known as the distribution axiom and (N) is also known as the necessitation rule. We use $\vdash_{\mathsf{K}} \varphi$ to denote the corresponding provability relation.

Theorem 2.7. For any modal logic formula φ, $\vDash_{\mathsf{K}} \varphi$ iff $\vdash_{\mathsf{K}} \varphi$.

2.8 MODAL μ-CALCULUS

Modal μ-calculus [24] is an extension of modal logic K with fixpoints.

Definition 2.28. The syntax of modal μ-calculus extends the syntax of modal logic K with an additional grammar rule:

$$\begin{array}{lll} \underline{\text{modal } \mu\text{-calculus formulas}} & \varphi ::= & (\text{syntax of modal logic K}) \\ & & \mid \mu X \,.\, \varphi \quad \text{if } \varphi \text{ is positive in } X \end{array}$$

where $X \in \mathrm{AP}$ is also an atomic proposition. Following the convention, we use p for free atomic propositions and X when they are bound by μ.

The operator μ is the least fixpoint operator. Its dual is ν, which is the greatest fixpoint operator, given by $\nu X \,.\, \varphi \equiv \neg \mu X \,.\, \neg \varphi[\neg X / X]$.

Definition 2.29. The modal μ-calculus satisfaction relation $T, \rho, s \vDash_{L\mu} \varphi$ extends the modal logic satisfaction relation $\vDash_{\mathsf{K}}$ by adding a rule for μ. Firstly, we introduce the notation

$$|\varphi|^{L\mu}_{T,\rho} = \{ s \in S \mid T, \rho, s \vDash_{L\mu} \varphi \}$$

(Modal Logic K)	all proof rules in Figure 2.3
(Pre-Fixpoint)	$\varphi[\mu X\,.\,\varphi/X] \to \mu X.\,\varphi$
(Knaster Tarski)	$\dfrac{\varphi[\psi/X] \to \psi}{\mu X.\,\varphi \to \psi}$

Figure 2.4: Sound and Complete Proof System of Modal μ-Calculus

Then, we add the following rule for μ:

$$|\mu X.\,\varphi|^{L\mu}_{T,\rho} = \bigcap\{A \subseteq S \mid |\varphi|^{L\mu}_{T,\rho[A/X]} \subseteq A\}$$

The derived rule for ν is

$$|\nu X.\,\varphi|^{L\mu}_{T,\rho} = \bigcup\{A \subseteq S \mid A \subseteq |\varphi|^{L\mu}_{T,\rho[A/X]}\}$$

We write $\vDash_{L\mu} \varphi$ iff $|\varphi|^{L\mu}_{T,\rho} = S$ for all T and ρ, that is, $T, \rho, s \vDash_{L\mu} \varphi$ for all T, ρ, and s.

Modal μ-calculus has a sound and complete proof system, as shown in Figure 2.4. We use $\vdash_{L\mu} \varphi$ to denote the corresponding provability relation.

Theorem 2.8. For any modal μ-calculus formula φ, $\vDash_{L\mu} \varphi$ iff $\vdash_{L\mu} \varphi$.

2.9 TEMPORAL LOGICS

We review three temporal logics: infinite-trace linear temporal logic (infinite-trace LTL), finite-trace linear temporal logic (finite-trace LTL), and computation tree logic (CTL).

2.9.1 Infinite-trace LTL

Definition 2.30. Let AP be a set of atomic propositions. The syntax of infinite-trace LTL is given by the following grammar:

$$\begin{array}{lll} \underline{\text{infinite-trace LTL formulas}} & \varphi ::= p \in AP & \\ & \quad \mid \varphi_1 \land \varphi_2 & \\ & \quad \mid \neg\varphi & \\ & \quad \mid \circ\varphi & // \text{ “next } \varphi\text{”} \end{array}$$

$$\mid \varphi_1 \, U \, \varphi_2 \qquad\qquad // \text{ ``}\varphi_1 \text{ until } \varphi_2\text{''}$$

The other modal operators such as $\diamond\varphi$ ("eventually φ") and $\Box\varphi$ ("always φ") can be defined as follows:

$$\diamond \varphi \equiv \top \, U \, \varphi \qquad\qquad \Box\varphi \equiv \neg \diamond \neg\varphi$$

where $\top$ is a formula that always holds, which can be defined by $\top \equiv p \vee \neg p$.

The models of infinite-trace LTL are infinite traces over $\mathcal{P}(AP)$. We use $\alpha = \alpha_0\alpha_1\alpha_2\ldots$ to denote an infinite trace, where $\alpha_i \subseteq AP$ for every $i \geq 0$. We use $\alpha_{\geq i}$ to denote the suffix trace $\alpha_i\alpha_{i+1}\alpha_{i+2}\ldots$.

Definition 2.31. The infinite-trace LTL satisfaction relation $\alpha \vDash_{\mathsf{infLTL}} \varphi$ is defined as follows:

1. $\alpha \vDash_{\mathsf{infLTL}} p$ iff $p \in \alpha_0$ for every $p \in AP$;
2. $\alpha \vDash_{\mathsf{infLTL}} \varphi_1 \wedge \varphi_2$ iff $\alpha \vDash_{\mathsf{infLTL}} \varphi_1$ and $\alpha \vDash \varphi_2$;
3. $\alpha \vDash_{\mathsf{infLTL}} \neg\varphi$ iff $\alpha \nvDash_{\mathsf{infLTL}} \varphi$;
4. $\alpha \vDash_{\mathsf{infLTL}} \circ\varphi$ iff $\alpha_{\geq 1} \vDash_{\mathsf{infLTL}} \varphi$;
5. $\alpha \vDash_{\mathsf{infLTL}} \varphi_1 \, U \, \varphi_2$ iff there exists $j \geq 0$ such that $\alpha_{\geq j} \vDash_{\mathsf{infLTL}} \varphi_2$ and for every $i < j$, $\alpha_{\geq i} \vDash_{\mathsf{infLTL}} \varphi_1$.

We write $\vDash_{\mathsf{infLTL}} \varphi$ to mean $\alpha \vDash_{\mathsf{infLTL}} \varphi$ for all α.

Infinite-trace LTL has a sound and complete proof system, as shown in Figure 2.5. We use $\vdash_{\mathsf{infLTL}} \varphi$ to denote the corresponding provability relation.

Theorem 2.9. For any infinite-trace LTL formula φ, $\vDash_{\mathsf{infLTL}} \varphi$ iff $\vdash_{\mathsf{infLTL}} \varphi$.

2.9.2 Finite-trace LTL

Finite execution traces play an important role in program verification and monitoring. Unlike infinite-trace LTL, finite-trace LTL use finite traces as its models.

Definition 2.32. Let AP be a set of atomic propositions. The syntax of finite-trace LTL is given by the following grammar:

$$\underline{\text{finite-trace LTL formulas}} \quad \varphi ::= p \in AP$$

(Propositional Tautology)	φ, if φ is a propositional tautology
(Modus Ponens)	$\dfrac{\varphi_1 \quad \varphi_1 \to \varphi_2}{\varphi_2}$
(K$_\bigcirc$)	$\circ(\varphi_1 \to \varphi_2) \to (\circ\varphi_1 \to \circ\varphi_2)$
(N$_\bigcirc$)	$\dfrac{\varphi}{\circ\varphi}$
(K$_\square$)	$\square(\varphi_1 \to \varphi_2) \to (\square\varphi_1 \to \square\varphi_2)$
(N$_\square$)	$\dfrac{\varphi}{\square\varphi}$
(Fun)	$\circ\varphi \leftrightarrow \neg(\circ\neg\varphi)$
(U1)	$(\varphi_1 \, U \, \varphi_2) \to \diamond\varphi_2$
(U2)	$(\varphi_1 \, U \, \varphi_2) \leftrightarrow (\varphi_2 \vee (\varphi_1 \wedge \circ(\varphi_1 \, U \, \varphi_2)))$
(ind)	$\square(\varphi \to \circ\varphi) \to (\varphi \to \square\varphi)$

Figure 2.5: Sound and Complete Proof System of Infinite-Trace LTL

$$
\begin{array}{ll}
\mid \varphi_1 \wedge \varphi_2 & \\
\mid \neg\varphi & \\
\mid \circ\varphi & // \text{“next } \varphi\text{”} \\
\mid \varphi_1 \, W \, \varphi_2 & // \text{“}\varphi_1 \text{ weak-until } \varphi_2\text{”}
\end{array}
$$

Unlike $\varphi_1 \, U \, \varphi_2$ in infinite-trace LTL, $\varphi_1 \, W \, \varphi_2$ only requires that φ_1 remains true until φ_2 becomes true, but it does not require that φ_2 eventually becomes true. Therefore, it is possible that φ_2 remains false until the end of the trace.

Definition 2.33. The finite-trace LTL satisfaction relation $\alpha_0 \dots \alpha_n \vDash_{\mathsf{finLTL}} \varphi$ with $n \geq 0$ is defined as follows:

1. $\alpha_0 \dots \alpha_n \vDash_{\mathsf{finLTL}} p$ iff $p \in \alpha_0$ for every $p \in AP$;

2. $\alpha_0 \dots \alpha_n \vDash_{\mathsf{finLTL}} \varphi_1 \wedge \varphi_2$ iff $\alpha_0 \dots \alpha_n \vDash_{\mathsf{finLTL}} \varphi_1$ and $\alpha_0 \dots \alpha_n \vDash \varphi_2$;

3. $\alpha_0 \dots \alpha_n \vDash_{\mathsf{finLTL}} \neg\varphi$ iff $\alpha_0 \dots \alpha_n \nvDash_{\mathsf{finLTL}} \varphi$;

(PROPOSITIONAL TAUTOLOGY)	φ, if φ is a propositional tautology
(MODUS PONENS)	$\dfrac{\varphi_1 \quad \varphi_1 \to \varphi_2}{\varphi_2}$
($\mathrm{K}_{\bigcirc}$)	$\circ(\varphi_1 \to \varphi_2) \to (\circ\varphi_1 \to \circ\varphi_2)$
($\mathrm{N}_{\bigcirc}$)	$\dfrac{\varphi}{\circ\varphi}$
($\mathrm{K}_{\Box}$)	$\Box(\varphi_1 \to \varphi_2) \to (\Box\varphi_1 \to \Box\varphi_2)$
($\mathrm{N}_{\Box}$)	$\dfrac{\varphi}{\Box\varphi}$
($\neg\bigcirc$)	$\neg\circ\varphi \to \circ\neg\varphi$
(COIND)	$\dfrac{\circ\varphi \to \varphi}{\varphi}$
(FIX)	$(\varphi_1 \, W \, \varphi_2) \leftrightarrow (\varphi_2 \vee (\varphi_1 \wedge \circ(\varphi_1 \, W \, \varphi_2)))$

Figure 2.6: Sound and Complete Proof System of Finite-Trace LTL

4. $\alpha_0 \ldots \alpha_n \vDash_{\mathsf{finLTL}} \circ\varphi$ iff $n = 0$ or $\alpha_1 \ldots \alpha_n \vDash_{\mathsf{finLTL}} \varphi$;

5. $\alpha_0 \ldots \alpha_n \vDash_{\mathsf{finLTL}} \varphi_1 \, W \, \varphi_2$ iff one of the following holds:

   (a) for every $i \le n$, $\alpha_i \ldots \alpha_n \vDash_{\mathsf{finLTL}} \varphi_1$;

   (b) there is $j \le n$ such that $\alpha_j \ldots \alpha_n \vDash_{\mathsf{finLTL}} \varphi_2$ and for every $i < j$, $\alpha_i \ldots \alpha_n \vDash_{\mathsf{finLTL}} \varphi_1$.

We write $\vDash_{\mathsf{finLTL}} \varphi$ to mean that $\alpha_0 \ldots \alpha_n \vDash_{\mathsf{finLTL}} \varphi$ for all n and $\alpha_0 \ldots \alpha_n$.

Finite-trace LTL has a sound and complete proof system, as shown in Figure 2.6. We use $\vdash_{\mathsf{finLTL}} \varphi$ to denote its provability relation.

Theorem 2.10. For any finite-trace LTL formula φ, $\vDash_{\mathsf{finLTL}} \varphi$ iff $\vdash_{\mathsf{finLTL}} \varphi$.

2.9.3 CTL

CTL is a branching-time logic whose time model has a tree-like structure.

Definition 2.34. Let AP be a set of atomic propositions. The syntax of CTL is given by the following grammar:

$$
\begin{array}{lll}
\underline{\text{CTL formulas}} \quad \varphi ::= p \in AP & & \\
\qquad\qquad\qquad\quad \mid \varphi_1 \wedge \varphi_2 & & \\
\qquad\qquad\qquad\quad \mid \neg\varphi & & \\
\qquad\qquad\qquad\quad \mid \mathsf{AX}\varphi & & \text{// ``on all paths, next } \varphi\text{''} \\
\qquad\qquad\qquad\quad \mid \mathsf{EX}\varphi & & \text{// ``on one path, next } \varphi\text{''} \\
\qquad\qquad\qquad\quad \mid \varphi_1 \,\mathsf{AU}\, \varphi_2 & & \text{// ``on all paths, } \varphi_1 \text{ until } \varphi_2\text{''} \\
\qquad\qquad\qquad\quad \mid \varphi_1 \,\mathsf{EU}\, \varphi_2 & & \text{// ``on one path, } \varphi_1 \text{ until } \varphi_2\text{''}
\end{array}
$$

Other modal operators can be defined as follows:

$$
\begin{array}{rll}
\mathsf{EF}\varphi \equiv \top \,\mathsf{EU}\, \varphi & & \text{// ``on one path, eventually } \varphi\text{''} \\
\mathsf{AF}\varphi \equiv \top \,\mathsf{AU}\, \varphi & & \text{// ``on all paths, eventually } \varphi\text{''} \\
\mathsf{AG}\varphi \equiv \neg\mathsf{EF}\neg\varphi & & \text{// ``on all paths, always } \varphi\text{''} \\
\mathsf{EG}\varphi \equiv \neg\mathsf{AF}\neg\varphi & & \text{// ``on one path, always } \varphi\text{''}
\end{array}
$$

As we can see, every CTL operator consists of a path quantifier (A or E) and a trace quantifier (X, U, F, or G). The path quantifiers specify whether a property should hold on all paths (A) or one path (E). The trace quantifiers have the same meaning as their infinite-trace LTL counterparts, where X is "next", U is "until", F is "eventually", and G is "always".

CTL models are infinite trees over $\mathcal{P}(AP)$. Given an infinite tree τ, we use $\mathsf{root}(\tau) \subseteq AP$ to denote its root and $\tau \to_{\text{subtree}} \tau'$ to indicate that τ' is an immediate sub-tree of τ.

Definition 2.35. The CTL satisfaction relation $\tau \vDash_{\mathsf{CTL}} \varphi$ is defined as follows:

- $\tau \vDash_{\mathsf{CTL}} p$ iff $p \in \mathsf{root}(\tau)$ for every $p \in AP$;
- $\tau \vDash_{\mathsf{CTL}} \varphi_1 \wedge \varphi_2$ iff $\tau \vDash_{\mathsf{CTL}} \varphi_1$ and $\tau \vDash_{\mathsf{CTL}} \varphi_2$;
- $\tau \vDash_{\mathsf{CTL}} \neg\varphi$ iff $\tau \nvDash_{\mathsf{CTL}} \varphi$;
- $\tau \vDash_{\mathsf{CTL}} \mathsf{AX}\varphi$ iff for all τ' such that $\tau \to_{\text{subtree}} \tau'$, $\tau' \vDash_{\mathsf{CTL}} \varphi$;
- $\tau \vDash_{\mathsf{CTL}} \mathsf{EX}\varphi$ iff there exists τ' such that $\tau \to_{\text{subtree}} \tau'$ and $\tau' \vDash_{\mathsf{CTL}} \varphi$;
- $\tau \vDash_{\mathsf{CTL}} \varphi_1 \,\mathsf{AU}\, \varphi_2$ if for all $\tau_0, \tau_1, \ldots$ such that $\tau = \tau_0 \to_{\text{subtree}} \tau_1 \to_{\text{subtree}} \ldots$, there exists $i \geq 0$ such that $\tau_i \vDash_{\mathsf{CTL}} \varphi_2$ and for all $j < i$, $\tau_j \vDash_{\mathsf{CTL}} \varphi_1$;

(Propositional Tautology)	φ, if φ is a propositional tautology
(Modus Ponens)	$\dfrac{\varphi_1 \quad \varphi_1 \to \varphi_2}{\varphi_2}$
(CTL$_1$)	$\mathsf{EX}(\varphi_1 \lor \varphi_2) \leftrightarrow \mathsf{EX}\varphi_1 \lor \mathsf{EX}\varphi_2$
(CTL$_2$)	$\mathsf{AX}\varphi \leftrightarrow \neg(\mathsf{EX}\neg\varphi)$
(CTL$_3$)	$\varphi_1\ \mathsf{EU}\ \varphi_2 \leftrightarrow \varphi_2 \lor (\varphi_1 \land \mathsf{EX}(\varphi_1\ \mathsf{EU}\ \varphi_2))$
(CTL$_4$)	$\varphi_1\ \mathsf{AU}\ \varphi_2 \leftrightarrow \varphi_2 \lor (\varphi_1 \land \mathsf{AX}(\varphi_1\ \mathsf{AU}\ \varphi_2))$
(CTL$_5$)	$\mathsf{EX}\top \land \mathsf{AX}\top$
(CTL$_6$)	$\mathsf{AG}(\varphi_3 \to (\neg\varphi_2 \land \mathsf{EX}\varphi_3)) \to (\varphi_3 \to \neg(\varphi_1\ \mathsf{AU}\ \varphi_2))$
(CTL$_7$)	$\mathsf{AG}(\varphi_3 \to (\neg\varphi_2 \land (\varphi_1 \to \mathsf{AX}\varphi_3))) \to (\varphi_3 \to \neg(\varphi_1\ \mathsf{EU}\ \varphi_2))$
(CTL$_8$)	$\mathsf{AG}(\varphi_1 \to \varphi_2) \to (\mathsf{EX}\varphi_1 \to \mathsf{EX}\varphi_2)$

Figure 2.7: Sound and Complete Proof System of CTL

- $\tau \vDash_{\mathsf{CTL}} \varphi_1\ \mathsf{EU}\ \varphi_2$ iff there exists $\tau_0, \tau_1, \ldots$ such that $\tau = \tau_0 \to_{\text{subtree}} \tau_1 \to_{\text{subtree}} \ldots$, and there exists $i \geq 0$ such that $\tau_i \vDash_{\mathsf{CTL}} \varphi_2$ and for all $j < i$, $\tau_j \vDash_{\mathsf{CTL}} \varphi_1$.

We write $\vDash_{\mathsf{CTL}} \varphi$ iff $\tau \vDash_{\mathsf{CTL}} \varphi$ for all τ.

CTL has a sound and complete proof system, as shown in Figure 2.7. We use $\vdash_{\mathsf{CTL}} \varphi$ to denote the corresponding provability relation.

Theorem 2.11. For any CTL formula φ, $\vDash_{\mathsf{CTL}} \varphi$ iff $\vdash_{\mathsf{CTL}} \varphi$.

2.10 DYNAMIC LOGIC

Dynamic logic, abbreviated as DL, is a common logic for program reasoning [12, 25, 26, 27].

Definition 2.36. Let AP be a set of atomic propositions and $APgm$ be a set of *atomic programs*. The syntax of DL is given by the following grammar:

$$\begin{aligned} \underline{\text{DL formulas}} \quad \varphi ::= {} & p \in AP \\ & \mid \varphi_1 \land \varphi_2 \end{aligned}$$

$$\begin{array}{llll} & & \mid \neg\varphi & \\ & & \mid [\alpha]\varphi & \\ \underline{\text{DL programs}} & \alpha ::= & a \in \mathit{APgm} & \\ & & \mid \alpha_1 \mathbin{;} \alpha_2 & \text{// sequence} \\ & & \mid \alpha_1 \cup \alpha_2 & \text{// choice} \\ & & \mid \alpha^* & \text{// iteration} \\ & & \mid \varphi? & \text{// test} \end{array}$$

The dual of $[\alpha]$ is $\langle\alpha\rangle$, given by $\langle\alpha\rangle\varphi \equiv \neg[\alpha](\neg\varphi)$. Other program constructs such as if-then-else, while-do, etc., can be defined as derived constructs [25, 26, 27].

Definition 2.37. Given an atomic program set *APgm*, a *DL model* is a tuple $T = (S, \{\xrightarrow{a}\}_{a\in \mathit{APgm}})$ where S is a set of states and $\xrightarrow{a} \subseteq S \times S$ is a transition relation for every $a \in \mathit{APgm}$. A *DL T-valuation* is a function $\rho\colon \mathit{AP} \to \mathcal{P}(S)$. The DL satisfaction relation $T, \rho, s \vDash_{\mathsf{DL}} \varphi$ is defined as follows. Firstly, we introduce the notation

$$|\varphi|^{\mathsf{DL}}_{T,\rho} = \{s \in S \mid T, \rho, s \vDash_{\mathsf{DL}} \varphi\}$$

Then, we define $|\varphi|^{\mathsf{DL}}_{T,\rho} \subseteq S$ and $|\alpha|^{\mathsf{DL}}_{T,\rho} \subseteq S \times S$ for all φ, α, and ρ using the following rules:

1. $|p|^{\mathsf{DL}}_{T,\rho} = \rho(p)$ for $p \in \mathit{AP}$;
2. $|\varphi_1 \wedge \varphi_2|^{\mathsf{DL}}_{T,\rho} = |\varphi_1|^{\mathsf{DL}}_{T,\rho} \cap |\varphi_2|^{\mathsf{DL}}_{T,\rho}$;
3. $|\neg\varphi|^{\mathsf{DL}}_{T,\rho} = S \setminus |\varphi|^{\mathsf{DL}}_{T,\rho}$;
4. $|[\alpha]\varphi|^{\mathsf{DL}}_{T,\rho} = \{s \in S \mid \text{for all } t \in S,\ (s,t) \in |\alpha|^{\mathsf{DL}}_{T,\rho} \text{ implies } t \in |\varphi|^{\mathsf{DL}}_{T,\rho}\}$;
5. $|a|^{\mathsf{DL}}_{T,\rho} = (\xrightarrow{a})$ for $a \in \mathit{APgm}$;
6. $|\alpha_1 \mathbin{;} \alpha_2|^{\mathsf{DL}}_{T,\rho} = |\alpha_1|^{\mathsf{DL}}_{T,\rho} \circ |\alpha_2|^{\mathsf{DL}}_{T,\rho}$;
7. $|\alpha_1 \cup \alpha_2|^{\mathsf{DL}}_{T,\rho} = |\alpha_1|^{\mathsf{DL}}_{T,\rho} \cup |\varphi_2|^{\mathsf{DL}}_{T,\rho}$;
8. $|\alpha^*|^{\mathsf{DL}}_{T,\rho} = (|\alpha|^{\mathsf{DL}}_{T,\rho})^*$;
9. $|\varphi?|^{\mathsf{DL}}_{T,\rho} = \{(s,s) \mid s \in |\varphi|^{\mathsf{DL}}_{T,\rho}\}$.

Recall that $|\alpha_1|^{\mathsf{DL}}_{T,\rho} \circ |\alpha_2|^{\mathsf{DL}}_{T,\rho}$ is the composition of $|\alpha_1|^{\mathsf{DL}}_{T,\rho}$ and $|\alpha_1|^{\mathsf{DL}}_{T,\rho}$, and $(|\alpha|^{\mathsf{DL}}_{T,\rho})^*$ is the reflexive and transitive closure of $|\alpha|^{\mathsf{DL}}_{T,\rho}$, as defined in Section 2.1. We write $\vDash_{\mathsf{DL}} \varphi$ iff $|\varphi|^{\mathsf{DL}}_{T,\rho} = S$ for all T and ρ.

(Propositional Tautology)	φ, if φ is a propositional tautology
(Modus Ponens)	$\dfrac{\varphi_1 \quad \varphi_1 \to \varphi_2}{\varphi_2}$
(DL$_1$)	$[\alpha](\varphi_1 \to \varphi_2) \to ([\alpha]\varphi_1 \to [\alpha]\varphi_2)$
(DL$_2$)	$[\alpha](\varphi_1 \wedge \varphi_2) \leftrightarrow ([\alpha]\varphi_1 \wedge [\alpha]\varphi_2)$
(DL$_3$)	$[\alpha \cup \beta]\varphi \leftrightarrow [\alpha]\varphi \wedge [\beta]\varphi$
(DL$_4$)	$[\alpha \,;\, \beta]\varphi \leftrightarrow [\alpha][\beta]\varphi$
(DL$_5$)	$[\psi?]\varphi \leftrightarrow (\psi \to \varphi)$
(DL$_6$)	$\varphi \wedge [\alpha][\alpha^*]\varphi \leftrightarrow [\alpha^*]\varphi$
(DL$_7$)	$\varphi \wedge [\alpha^*](\varphi \to [\alpha]\varphi) \to [\alpha^*]\varphi$
(Gen)	$\dfrac{\varphi}{[\alpha]\varphi}$

Figure 2.8: Sound and Complete Proof System of DL

DL has sound and complete proof system, as shown in Figure 2.8. We use $\vdash_{\mathsf{DL}} \varphi$ to denote the corresponding provability relation.

Theorem 2.12. For any DL formula φ, $\vDash_{\mathsf{DL}} \varphi$ iff $\vdash_{\mathsf{DL}} \varphi$.

2.11 λ-CALCULUS

λ-calculus [28] is a Turing-complete foundation of computation based on function abstraction and application.

Definition 2.38. Let V be a set of variables denoted by x, y, etc. The syntax of λ-calculus is as follows:

$$\begin{array}{lll}
\underline{\lambda\text{-expressions}} & e ::= x & \\
& \quad\mid e_1\, e_2 & \text{// function application} \\
& \quad\mid \lambda x \,.\, e & \text{// function abstraction (i.e., } \lambda\text{-abstraction}
\end{array}$$

where λ is a binder. We use $\mathit{freeVar}(e) \subseteq V$ to denote the free variables of e and $e[e'/x]$ to denote the result of substituting e' for x in e, where α-renaming implicitly happens to avoid variable capture. Let Λ be the set of all λ-expressions.

In λ-calculus, we are interested in proving equations between λ-expressions. Equational reasoning in λ-calculus includes the standard reflexivity, symmetry, transitivity, and congruence proof rules in Figure 2.2, plus a distinguished (β) axiom schema that specifies the result of function application:

$$(\beta) \quad (\lambda x \,.\, e)\, e' = e[e'/x] \quad \text{for all } x \in V \text{ and } e, e' \in \Lambda$$

We write $\vdash_\lambda e_1 = e_2$ to mean that $e_1 = e_2$ is provable.

λ-calculus has many notions of models. Here, we review the concrete Cartesian closed category models, abbreviated as concrete ccc models (see [29, Definition 5.5.9]).

Firstly, we define application structures.

Definition 2.39. An *application structure* is a tuple $(A, \_ \bullet_A \_)$ where A is a set and $\_ \bullet_A \_ \colon A \times A \to A$ is a binary function.

Next, we define pre-models.

Definition 2.40. Given an applicative structure $(A, \_ \bullet_A \_)$ and $a \in A$, we define a function $\mathbb{A}(a) \colon A \to A$ given by $\mathbb{A}(a)(b) = a \bullet_A b$ for all $b \in A$. Let $R(A) = \mathsf{codomain}(\mathbb{A})$, which is called the set of *representable functions*; that is,

$$R(A) = \{f \colon A \to A \mid \text{there is } a \in A \text{ such that } f = \mathbb{A}(a)\}$$

If there exists $\mathcal{G} \colon R(A) \to A$ such that $\mathbb{A} \circ \mathcal{G}$ is the identity function over $R(A)$, we call $\mathcal{G}$ a *retraction function*, and $(A, \_ \bullet_A \_, \mathcal{G})$ a *pre-model.*

Finally, we define concrete ccc models.

Definition 2.41. A *concrete ccc model* is a pre-model in Definition 2.40, if the following rules for defining $|e|^\lambda_\rho$ for all $\rho \colon V \to A$ are well-defined:

1. $|x|^\lambda_\rho = \rho(x)$;
2. $|e_1 e_2|^\lambda_\rho = |e_1|^\lambda_\rho \bullet_A |e_2|^\lambda_\rho$;
3. $|\lambda x \,.\, e|^\lambda_\rho = \mathcal{G}(f^\rho_{e,x})$ where $f^\rho_{e,x}(a) = |e|^\lambda_{\rho[a/x]}$ for $a \in A$ and $f^\rho_{e,x} \in R(A)$.

We write $A \vDash_\lambda e_1 = e_2$ iff $|e_1|^\lambda_\rho = |e_2|^\lambda_\rho$ for all ρ. We write $\vDash_\lambda e_1 = e_2$ iff $A \vDash_\lambda e_1 = e_2$ for all concrete ccc models A.

Concrete ccc models are sound and complete with respect to $\vdash_\lambda$, which denotes equational reasoning over λ-expressions.

Theorem 2.13. For any λ-expressions e_1 and e_2, $\vDash_\lambda e_1 = e_2$ iff $\vdash_\lambda e_1 = e_2$.

2.12 TERM-GENERIC FIRST-ORDER LOGIC

Term-generic first-order logic [30], or simply term-generic logic (abbreviated as TGL), is a variant of many-sorted FOL whose syntax is parametric in a set of generic terms. Generic terms are generalization of FOL terms in Definition 2.3, which are inductively built using function symbols. Generic terms are not built using function symbols, but defined axiomatically. There are two operations related to generic terms: free variables $\mathit{freeVar}(e)$ and capture-avoiding substitution $e[e'/x]$. These operations should satisfy certain conditions [30, Definition 2.1]. TGL formulas are built in the same way as FOL formulas, except that FOL terms are now replaced by generic terms.

TGL aims at defining various logics and calculi that feature bindings, such as λ-calculus. These bindings-featuring systems usually cannot be naturally defined as FOL theories. On the other hand, the generic terms in TGL can be instantiated to different kinds of concrete terms, such as the expressions of λ-calculus. This way, many bindings-featuring systems can be naturally defined as TGL theories.

In this work, we do not need TGL in its full generality. Instead, we present a concrete instance of TGL where the generic terms are inductively built from a syntax that features binders of the form $b(x\,{:}\,s_1, t_{s_2})$, where $x\,{:}\,s_1$ is bound by b in t_{s_2}. The semantics and proof system of TGL will also be introduced using this concrete instance.

Definition 2.42. A *(many-sorted) binder signature* is a tuple (S, F, B, Π), where (S, F, Π) is a FOL signature and $B = \{B_{s_1,s_2,s}\}_{s_1,s_2,s \in S}$ is an S^3-indexed set of *binders*. Let $V = \{V_s\}_{s \in S}$ be an S-indexed set of variables. The syntax of TGL is given by the following grammar:

$$\begin{array}{rl}
\underline{\text{TGL } (S,F,B,\Pi)\text{-terms}} & t_s ::= \text{(syntax of FOL terms)} \\
& \quad\;\; \mid b(x\,{:}\,s_1, t_{s_2}) \;\; \text{with } b \in B_{s_1,s_2,s} \\
\underline{\text{TGL formulas}} & \varphi ::= \text{(syntax of FOL formulas)} \\
& \quad\;\; \mid t_s = t'_s
\end{array}$$

We use TGLTERM and TGLFORM to denote the sets of TGL terms and formulas, respectively.

Unlike FOL, the semantics of TGL has a Henkin-style definition, where terms and formulas are interpreted at the same time.

Definition 2.43. Let $A = \{A_s\}_{s \in S}$ be an S-indexed set. A *TGL A-valuation* or simply *TGL valuation* is a function $\rho \colon V \to A$. Let TGLVAL $= [V \to A]$ be the set of all TGL valuations. A *TGL model* is a tuple $(\{A_s\}_{s \in S}, \{A_t\}_{t \in \text{TGLTERM}}, \{A_\pi\}_{\pi \in \Pi})$, where

1. $A_{t_s} \colon \text{TGLVAL} \to A_s$ is a function for every $t_s \in \text{TGLTERM}$; in addition, the following conditions should hold for all $x : s \in V_s$, $t_s, t'_s \in \text{TGLTERM}$, and $\rho \in \text{TGLVAL}$:
   (a) $A_{x:s}(\rho) = \rho(x : s)$.
   (b) $A_{t_s[t'_s/x:s]}(\rho) = A_{t_s}(\rho[A_{t'_s}(\rho)/x : s])$;
2. $A_\pi \subseteq A_{s_1} \times \cdots \times A_{s_n}$ for every $\pi \in \Pi_{s_1 \ldots s_n}$.

Definition 2.44. Under the notation of Definition 2.43, we define $A_\varphi \subseteq \text{TGLVAL}$ for every $\varphi \in \text{TGLFORM}$ using the following rules:

1. $\rho \in A_{t_s = t'_s}$ iff $A_{t_s}(\rho) = A_{t'_s}(\rho)$;
2. $\rho \in A_{\pi(t_{s_1}, \ldots, t_{s_n})}$ iff $A_\pi(A_{t_{s_1}}(\rho), \ldots, A_{t_{s_n}}(\rho))$ holds;
3. $\rho \in A_{\varphi_1 \wedge \varphi_2}$ iff $\rho \in A_{\varphi_1}$ and $\rho \in A_{\varphi_2}$;
4. $\rho \in A_{\neg\varphi}$ iff $\rho \notin A_\varphi$;
5. $\rho \in A_{\exists x : s \,.\, \varphi}$ iff there exists $a \in A_s$ such that $\rho[a/x : s] \in A_\varphi$.

The TGL satisfaction relation $A, \rho \vDash_{\mathsf{TGL}} \varphi$ is defined by $\rho \in A_\varphi$. We write $A \vDash_{\mathsf{TGL}} \varphi$ iff $A, \rho \vDash_{\mathsf{TGL}} \varphi$ for all ρ, that is, $A_\varphi = \text{TGLVAL}$. Given a set Γ of TGL formulas, we write $A \vDash_{\mathsf{TGL}} \Gamma$ iff $A \vDash_{\mathsf{TGL}} \psi$ for all $\psi \in \Gamma$. For two sets Δ_1 and Δ_2, we write $\Gamma \vDash_{\mathsf{TGL}} \Delta_1 \triangleright \Delta_2$ iff $\bigcap_{\varphi \in \Delta_1} A_\varphi \subseteq \bigcup_{\varphi \in \Delta_2} A_\varphi$ for all $A \vDash_{\mathsf{TGL}} \Gamma$. Intuitively, $\Delta_1 \triangleright \Delta_2$ states that if all the formulas in Δ_1 hold, then one of the formulas in Δ_2 holds.

TGL has a sound and complete Gentzen-style proof system, as shown in Figure 2.9. The proof system derives sequents of the form $\Gamma \vdash_{\mathsf{TGL}} \Delta_1 \triangleright \Delta_2$, where $\Gamma, \Delta_1, \Delta_2 \subseteq \text{TGLFORM}$. Following the convention of writing Gentzen-style proof rules, we write Δ, φ to mean $\Delta \cup \{\varphi\}$. We require that all the formulas in Γ are closed (i.e.,) and Δ_1, Δ_2 are finite. These requirements are needed for Theorem 2.14.

Theorem 2.14 ([30, Theorem 3.1])**.** Let Γ be a set of closed TGL formulas. For any finite $\Delta_1, \Delta_2 \in \text{TGLFORM}$, $\Gamma \vDash_{\mathsf{TGL}} \Delta_1 \triangleright \Delta_2$ iff $\Gamma \vdash_{\mathsf{TGL}} \Delta_1 \triangleright \Delta_2$.

(Ax)
$$\frac{\cdot}{\Delta_1 \rhd \Delta_2} \text{ if } \Delta_1 \cap \Delta_2 \neq \emptyset$$

(Left→)
$$\frac{\Delta_1 \rhd \Delta_2, \varphi_1 \quad \Delta_1, \varphi_2 \rhd \Delta_2}{\Delta_1, (\varphi_1 \to \varphi_2) \rhd \Delta_2}$$

(Right→)
$$\frac{\Delta_1, \varphi \rhd \Delta_2, \varphi_2}{\Delta_1 \rhd \Delta_2, (\varphi_1 \to \varphi_2)}$$

(Left∧)
$$\frac{\Delta_1, \varphi_1, \varphi_2 \rhd \Delta_2}{\Delta_1, (\varphi_1 \wedge \varphi_2) \rhd \Delta_2}$$

(Right∧)
$$\frac{\Delta_1 \rhd \Delta_2, \varphi_1 \quad \Delta_1 \rhd \Delta_2, \varphi_2}{\Delta_1 \rhd \Delta_2, (\varphi_1 \wedge \varphi_2)}$$

(Left∀)
$$\frac{\Delta_1, \forall x \,.\, \varphi, \varphi[t/x] \rhd \Delta_2}{\Delta_1, \forall x \,.\, \varphi \rhd \Delta_2}$$

(Right∀)
$$\frac{\Delta_1 \rhd \Delta_2, \varphi[y/x]}{\Delta_1 \rhd \Delta_2, \forall x \,.\, \varphi} \text{ if } y \text{ is fresh}$$

(Reflexivity)
$$\frac{\Delta_1, t = t \rhd \Delta_2}{\Delta_1 \rhd \Delta_2}$$

(Symmetry)
$$\frac{\Delta_1 \rhd \Delta_2, t_1 = t_2 \quad \Delta_1, t_2 = t_1 \rhd \Delta_2}{\Delta_1 \rhd \Delta_2}$$

(Transitivity)
$$\frac{\Delta_1 \rhd \Delta_2, t_1 = t_2 \quad \Delta_1 \rhd \Delta_2, t_2 = t_3 \quad \Delta_1, t_1 = t_3 \rhd \Delta_2}{\Delta_1 \rhd \Delta_2}$$

(Cmp$_\pi$)
$$\frac{\begin{array}{c} \Delta_1 \rhd \Delta_2, t_i = t_i' \text{ for all } 1 \leq i \leq n \\ \Delta_1 \rhd \Delta_2, \pi(t_1, \ldots, t_n) \\ \Delta_1, \pi(t_1', \ldots, t_n') \rhd \Delta_2 \end{array}}{\Delta_1 \rhd \Delta_2}$$

(Sbs)
$$\frac{\Delta_1 \rhd \Delta_2, t_1 = t_2 \quad \Delta_1, t[t_1/x] = t[t_2/x] \rhd \Delta_2}{\Delta_1 \rhd \Delta_2}$$

(Binder)
$$\frac{\Delta_1 \rhd \Delta_2, t = t' \quad \Delta_1, b(x, t) = b(x, t') \rhd \Delta_2}{\Delta_1 \rhd \Delta_2}$$

Figure 2.9: Sound and Complete Proof System of TGL [30, Figs. 1-2] Plus (Binder)

2.13 MATCHING LOGIC

Matching logic [2] is a variant of many-sorted FOL that makes no distinction between function and predicate symbols, allowing them to uniformly build patterns. Patterns define both structural and logical constraints and are interpreted in models as sets of elements, that is, those that match them.

2.13.1 Matching logic syntax and semantics

Definition 2.45. A *matching logic signature* (S, Σ) is the same as a many-sorted signature, where we call the elements in Σ *matching logic symbols* or simply *symbols*. Given a matching logic signature (S, Σ) and an S-indexed set $V = \{V_s\}_{s \in S}$ of variables denoted by $x : s$, $y : s$, etc., the syntax of matching logic is given by the following grammar:

$$\begin{array}{lll} \underline{\text{matching logic patterns}} & \varphi_s ::= x : s \in V_s & \\ & \quad \mid \sigma(\varphi_{s_1}, \ldots, \varphi_{s_n}) & \text{with } \sigma \in \Sigma_{s_1 \ldots s_n, s} \\ & \quad \mid \varphi_s \wedge \varphi'_s & \\ & \quad \mid \neg \varphi_s & \\ & \quad \mid \exists x : s' \,.\, \varphi_s & \end{array}$$

Let $\text{MLP}\textsc{attern}(S, V, \Sigma) = \{\text{MLP}\textsc{attern}_s(S, V, \Sigma)\}_{s \in S}$ be the S-indexed set of patterns. We feel free to drop the parameters S, V, and even Σ when they are understood or not important.

We adopt common abbreviation and shortcuts whenever possible. We feel free to drop the sorts. For example, we write x and φ instead of $x : s$ and φ_s when s is not important. When we write a pattern, we assume it is well-formed and well-sorted, without explicitly specifying the necessary conditions. For example, when we write $\varphi_1 \to \varphi_2$, it is understood that φ_1 and φ_2 should have the same sort. When we write $\sigma(\varphi_1, \ldots, \varphi_n)$, it is understood that $\varphi_1, \ldots, \varphi_n$ should have the corresponding argument sorts as σ. When $n = 0$, we call σ a *constant symbol* and write $\sigma \in \Sigma_{\epsilon, s}$. We write σ to mean the pattern $\sigma()$. We define the following notation:

$$\begin{array}{ll} \varphi_1 \vee \varphi_2 \equiv \neg(\neg \varphi_1 \wedge \neg \varphi_2) & \forall x : s \,.\, \varphi \equiv \neg \exists x : s \,.\, \neg \varphi \\ \varphi_1 \to \varphi_2 \equiv \neg \varphi_1 \vee \varphi_2 & \top_s \equiv \exists x : s \,.\, x : s \\ \varphi_1 \leftrightarrow \varphi_2 \equiv (\varphi_1 \to \varphi_2) \wedge (\varphi_2 \to \varphi_1) & \bot_s \equiv \neg \top_s \end{array}$$

The only non-trivial definition is $\top_s \equiv \exists x : s \,.\, x : s$. Its correctness is shown in Proposition 2.15.

Like in FOL, $\exists$ and $\forall$ are binders. We adopt the standard notions of free variables, α-renaming, and capture-avoiding substitution. We let $\mathit{freeVar}(\varphi)$ denote the set of free variables in φ. When $\mathit{freeVar}(\varphi) = \emptyset$, we say φ is closed. We regard α-equivalent patterns φ and φ' as the same, and write $\varphi \equiv \varphi'$. We let $\varphi[\psi/x]$ be the result of substituting ψ for every free occurrence of x in φ, where α-renaming happens implicitly to prevent variable capture. We let $\varphi[\psi_1/x_1, \ldots, \psi_n/x_n]$ be the result of simultaneously substituting $\psi_1, \ldots, \psi_n$ for $x_1, \ldots, x_n$.

Definition 2.46. Given a matching logic signature (S, Σ), a *matching logic* (S, Σ)*-model* or simply a *model* is a tuple $M = (\{M_s\}_{s \in S}, \{\sigma_M\}_{\sigma \in \Sigma})$, where

- M_s is a nonempty carrier set, for every $s \in S$;
- $\sigma_M \colon M_{s_1} \times \cdots \times M_{s_n} \to \mathcal{P}(M_s)$ is a function, for every $\sigma \in \Sigma_{s_1 \ldots s_n, s}$.

FOL function symbols can be regarded as a special instance of matching logic symbols, where $\mathsf{card}(\sigma_M(a_1, \ldots, a_n)) = 1$ for all $a_1 \in M_{s_1}, \ldots, a_n \in M_{s_n}$. Similarly, partial functions in partial FOL [31] can be regarded as a special instance, where $\mathsf{card}(\sigma_M(a_1, \ldots, a_n)) \geq 1$ for all $a_1 \in M_{s_1}, \ldots, a_n \in M_{s_n}$. The undefinedness of σ_M at $a_1, \ldots, a_n$ is captured by letting $\sigma_M(a_1, \ldots, a_n) = \emptyset$.

Definition 2.47. Given a matching logic (S, Σ) and an (S, Σ)-model M. An *M-valuation* or simply *valuation* is a function $\rho \colon V \to M$. The matching logic interpretation function $|\_|_{M,\rho} \colon \textsc{MLPattern} \to \mathcal{P}(M)$ is inductively defined as follows:

- $|x : s|_{M,\rho} = \{\rho(x : s)\}$ for all $x : s \in V_s$;
- $|\varphi_1 \wedge \varphi_2|_{M,\rho} = |\varphi_1|_{M,\rho} \cap |\varphi_2|_{M,\rho}$;
- $|\neg\varphi|_{M,\rho} = M_s \setminus |\varphi|_{M,\rho}$ for every $\varphi_s \in \textsc{MLPattern}_s$;
- $|\exists x \,.\, \varphi|_{M,\rho} = \bigcup_{a \in M_{s'}} |\varphi|_{M,\rho[a/x]}$;
- $|\sigma(\varphi_1, ..., \varphi_n)|_{M,\rho} = \sigma_M^{\text{ext}}(|\varphi_1|_{M,\rho}, ..., |\varphi_n|_{M,\rho})$ for $\sigma \in \Sigma_{s_1 \ldots s_n, s}$;

where σ_M^{ext} is the pointwise extension of σ_M in Section 2.1. We feel free to drop the subscripts M and ρ when they are not important or relevant. We say that φ_s is *valid* in M, written $M \vDash \varphi_s$, iff $|\varphi_s|_{M,\rho} = M_s$ for all ρ.

Intuitively, $|\varphi|_{M,\rho}$ is the set of elements that match φ under M and ρ. There is a close relation between the semantics of patterns and set operations. For example, $\varphi_1 \wedge \varphi_2$ is matched

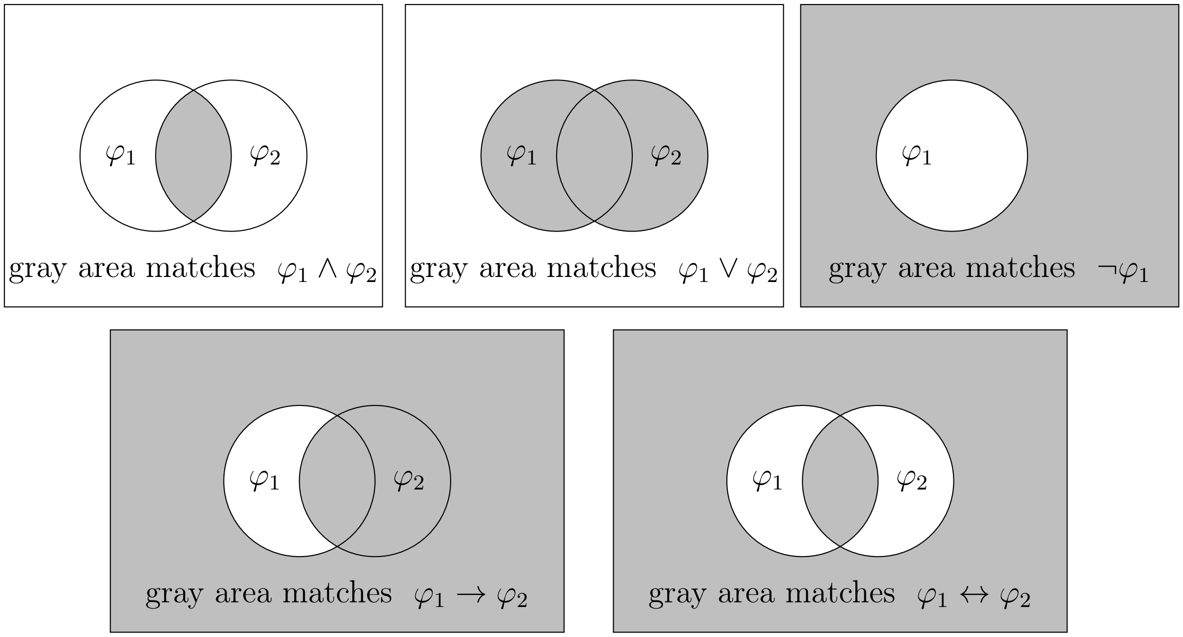

φ_1
φ_2
gray area matches $\varphi_1 \wedge \varphi_2$
φ_1
φ_2
gray area matches $\varphi_1 \vee \varphi_2$
φ_1
gray area matches $\neg\varphi_1$
φ_1
φ_2
gray area matches $\varphi_1 \to \varphi_2$
φ_1
φ_2
gray area matches $\varphi_1 \leftrightarrow \varphi_2$

Figure 2.10: Matching Logic Semantics Illustration [2, Fig. 4]

by those elements that match both φ_1 and φ_2. Therefore, $|\varphi_1 \wedge \varphi|_{M,\rho} = |\varphi_1|_{M,\rho} \cap |\varphi|_{M,\rho}$. In other words, conjunction ($\wedge$) means set intersection. Similarly, disjunction ($\vee$) means set union and negation ($\neg$) means set complement.

Proposition 2.15. $|\exists x : s \,.\, x : s|_{M,\rho} = M_s$.

Proof. By definition, $|x : s \,.\, x : s|_{M,\rho} = \bigcup_{a \in M_s} |x : s|_{M,\rho} = \bigcup_{a \in M_s} \{a\} = M_s$. QED.

Definition 2.48. Given a matching logic signature (S, Σ), a *matching logic* (S, Σ)*-theory* or simply *theory* is a tuple (S, Σ, Γ), where Γ is a set of (S, Σ)-patterns/axioms. When (S, Σ) is understood, we simply use Γ to denote the theory (S, Σ, Γ). We write $M \vDash \Gamma$ iff $M \vDash \psi$ for all $\psi \in \Gamma$. We write $\Gamma \vDash \varphi$ iff $M \vDash \varphi$ for all $M \vDash \Gamma$.

2.13.2 Important theories

Several mathematical instruments of practical importance, such as definedness, equality, membership, and functions, can be defined as matching logic theories.

Definition 2.49. For a set S of sorts, let $\Sigma^{\mathsf{definedness}}$ and $\Gamma^{\mathsf{definedness}}$ be as follows:

$$\Sigma^{\mathsf{definedness}} = \{\lceil \_ \rceil_s^{s'} \mid s, s' \in S\} \qquad \text{// definedness symbols}$$

$$\Gamma^{\mathsf{definedness}} = \{\lceil x : s \rceil_s^{s'} \mid s, s' \in S\} \qquad \text{// (D}\textsc{efinedness}\text{) axioms}$$

Furthermore, we introduce the following notation:

$$
\begin{aligned}
\lfloor \varphi \rfloor_s^{s'} &\equiv \neg \lceil \neg \varphi \rceil_s^{s'} && \text{// totality} \\
\varphi_1 =_s^{s'} \varphi_2 &\equiv \lfloor \varphi_1 \leftrightarrow \varphi_2 \rfloor_s^{s'} && \text{// equality} \\
x \in_s^{s'} \varphi &\equiv \lceil x \wedge \varphi \rceil_s^{s'} && \text{// membership} \\
\varphi_1 \subseteq_s^{s'} \varphi_2 &\equiv \lfloor \varphi_1 \to \varphi_2 \rfloor_s^{s'} && \text{// set inclusion}
\end{aligned}
$$

and feel free to drop the sort superscripts/subscripts when they are not important.

Proposition 2.16. *For any $M \vDash \Gamma^{\mathsf{definedness}}$, the following hold:*

- $(\lceil \_ \rceil_s^{s'})_M(a) = M_{s'}$ *for all* $a \in M_s$;
- $|\lceil \varphi \rceil_s^{s'}|_{M,\rho} = M_{s'}$ *if* $|\varphi|_{M,\rho} \neq \emptyset$; *otherwise,* $|\lceil \varphi \rceil_s^{s'}|_{M,\rho} = \emptyset$;
- $|\lfloor \varphi \rfloor_s^{s'}|_{M,\rho} = M_{s'}$ *if* $|\varphi|_{M,\rho} = M_s$; *otherwise,* $|\lfloor \varphi \rfloor_s^{s'}|_{M,\rho} = \emptyset$;
- $|\varphi_1 =_s^{s'} \varphi_2|_{M,\rho} = M_{s'}$ *if* $|\varphi_1|_{M,\rho} = |\varphi_2|_{M,\rho}$; *otherwise,* $|\varphi_1 =_s^{s'} \varphi_2|_{M,\rho} = \emptyset$;
- $|x \in_s^{s'} \varphi|_{M,\rho} = M_{s'}$ *if* $\rho(x) \in |\varphi|_{M,\rho}$; *otherwise,* $|x \in_s^{s'} \varphi|_{M,\rho} = \emptyset$;
- $|\varphi_1 \subseteq_s^{s'} \varphi_2|_{M,\rho} = M_{s'}$ *if* $|\varphi_1|_{M,\rho} \subseteq |\varphi_1|_{M,\rho}$; *otherwise,* $|\varphi_1 \subseteq_s^{s'} \varphi_2|_{M,\rho} = \emptyset$; *in particular,* $|x \subseteq_s^{s'} \varphi|_{M,\rho} = |x \in_s^{s'} \varphi|_{M,\rho}$;
- $M \vDash \varphi_1 =_s^{s'} \varphi_2$ *if and only if* $M \vDash \varphi_1 \leftrightarrow \varphi_2$;
- $M \vDash \varphi_1 \subseteq_s^{s'} \varphi_2$ *if and only if* $M \vDash \varphi_1 \to \varphi_2$.

In Section 2.13.1, we have shown that functions (or partial functions) can be regarded as a special instance of matching logic symbols where $\mathsf{card}(\sigma_M(a_1, \ldots, a_n)) = 1$ (or ≤ 1). We can enforce the function (or partial function) semantics of a matching logic symbol using patterns/axioms.

Definition 2.50. Given a matching logic symbol $\sigma \in \Sigma_{s_1 \ldots s_n, s}$, let $\Gamma^{\mathsf{function}(\sigma)}$ and $\Gamma^{\mathsf{pfunction}(\sigma)}$ be as follows:

$$
\begin{aligned}
\Gamma^{\mathsf{function}(\sigma)} &= \{\exists y : s \,.\, \sigma(x_1 : s_1, \ldots, x_n : s_n) = y : s\} && \text{// (Function) axiom} \\
\Gamma^{\mathsf{pfunction}(\sigma)} &= \{\exists y : s \,.\, \sigma(x_1 : s_1, \ldots, x_n : s_n) \subseteq y : s\} && \text{// (Partial Function) axiom}
\end{aligned}
$$

For brevity, we use $\sigma \colon s_1 \times \cdots \times s_n \to s$ and $\sigma \colon s_1 \times \cdots \times s_n \rightharpoonup s$ to denote (Function) and (Partial Function), respectively, and we feel free to drop the sorts when they are not important. Given $F \subseteq \Sigma$, which is a set of symbols to be interpreted as functions (or partial functions), we let $\Gamma^{\mathsf{function}(F)} = \bigcup_{f \in F} \Gamma^{\mathsf{function}(f)}$ (or $\Gamma^{\mathsf{pfunction}(F)} = \bigcup_{f \in F} \Gamma^{\mathsf{pfunction}(f)}$) to denote the corresponding set of axioms.

Unlike FOL where formulas are two-valued ($\top$ or $\bot$), matching logic patterns can be evaluated to any subsets of the underlying carrier sets. However, we can restore the FOL semantics by letting $\top_s$ denote logical truth and $\bot_s$ denote logical false in sort s. A FOL predicate symbol over $s_1 \dots s_n$ is a special instance of a matching logic symbol $\sigma \in \Sigma_{s_1 \dots s_n, \mathsf{Formula}}$ where $\mathsf{Formula}$ is a distinguished sort for formulas and $\sigma_M(a_1, \dots, a_n) \in \{\emptyset, M_{\mathsf{Formula}}\}$ for all $a_i \in M_{s_i}$, $1 \leq i \leq n$. Note that if M_{Formula} is a singleton set, the above condition is automatically satisfied for any symbol whose return sort is $\mathsf{Formula}$. Following this idea, we can define FOL in matching logic in the following way, which is a slightly modified version of the definition given in [2, Section 7].

Definition 2.51. Given a FOL signature (S, F, Π), we define the corresponding matching logic theory $(S^{\mathsf{FOL}(S,F,\Pi)}, \Sigma^{\mathsf{FOL}(S,F,\Pi)}, \Gamma^{\mathsf{FOL}(S,F,\Pi)})$, or simply $(S^{\mathsf{FOL}}, \Sigma^{\mathsf{FOL}}, \Gamma^{\mathsf{FOL}})$, as follows:

$$\begin{aligned} S^{\mathsf{FOL}} &= S \cup \{\mathsf{Formula}\} \\ \Sigma^{\mathsf{FOL}} &= \Gamma^{\mathsf{definedness}} \cup F \cup \{\pi \in \Sigma^{\mathsf{FOL}}_{s_1 \dots s_n, \mathsf{Formula}} \mid \pi \in \Pi_{s_1 \dots s_n}\} \\ \Gamma^{\mathsf{FOL}} &= \Gamma^{\mathsf{definedness}} \cup \Gamma^{\mathsf{function}(F)} \cup \{x : \mathsf{Formula}\} \end{aligned}$$

That is, we add all the FOL sorts to matching logic, with an additional sort $\mathsf{Formula}$ for FOL formulas. We include all the definedness symbols and axioms, which are necessary for defining functions. We add FOL function symbols as matching logic symbols of the same arities and define them using the function axioms. We add FOL predicate symbols as matching logic symbols with the return sort $\mathsf{Formula}$. The axiom $\{x : \mathsf{Formula}\}$ enforces the carrier set of $\mathsf{Formula}$ to be a singleton set, so we do not need any axioms for $\pi \in \Sigma^{\mathsf{FOL}}_{s_1 \dots s_n, \mathsf{Formula}}$.

This way, all FOL formulas are matching logic Σ^{FOL}-patterns of sort $\mathsf{Formula}$.

Proposition 2.17 ([2, Proposition 7.1])**.** *Under the notation of Definition 2.51,* $\vDash_{\mathsf{FOL}} \varphi$ *iff* $\Gamma^{\mathsf{FOL}} \vDash \varphi$ *for every FOL formula* φ*.*

Our definition of FOL in Definition 2.51 is different from [2, Section 7] in that we enforce $\mathsf{Formula}$ to be a singleton set using an additional axiom $\{x : \mathsf{Formula}\}$. Proposition 2.17 still holds, because for the "only if" direction, we only have fewer matching logic models. For the "if" direction, the original proof in [2] only needs matching logic models where the carrier set of $\mathsf{Formula}$ is a singleton set, anyway. We prefer Definition 2.51 because it is easier to extend it to other logics, such as LFP (see Section 5.2).

Another important result about the expressive power of matching logic is its ability to define separation logic (SL) as an instance, where we fix the underlying model to be the model of finite maps. However, the following result does not consider recursive symbols in SL, which will be discussed in Section 5.3.

Definition 2.52. We define the matching logic theory $(S^{\mathsf{Map}}, \Sigma^{\mathsf{Map}}, \Gamma^{\mathsf{Map}})$ for maps as follows:

$$\begin{aligned} S^{\mathsf{Map}} &= \{\mathsf{Nat}, \mathsf{Map}\} \\ \Sigma^{\mathsf{Map}} &= \Sigma^{\mathsf{definedness}} \cup \{\mathsf{nil}, \mathsf{emp}, (\_{\mapsto}\_), (\_{*}\_)\} \end{aligned}$$

and Γ^{Map} includes $\Gamma^{\mathsf{definedness}}$ plus the following axioms:

$$\begin{aligned} &\mathsf{nil} \colon \; \to \mathsf{Nat} \\ &\mathsf{emp} \colon \; \to \mathsf{Map} \\ &\_{\mapsto}\_ \colon \mathsf{Nat} \times \mathsf{Nat} \rightharpoonup \mathsf{Map} \\ &\_{*}\_ \colon \mathsf{Map} \times \mathsf{Map} \rightharpoonup \mathsf{Map} \\ &\mathsf{emp} * h = h \\ &h_1 * h_2 = h_2 * h_1 \\ &(h_1 * h_2) * h_3 = h_1 * (h_2 * h_3) \\ &\mathsf{nil} \mapsto x = \bot \\ &x \mapsto y * x \mapsto z = \bot \end{aligned}$$

Furthermore, we define $\varphi_1 \mathrel{-\!\!*} \varphi_2 \equiv \exists h \,.\, h \wedge \lfloor h * \varphi_1 \to \varphi_2 \rfloor$.

This way, all SL formulas (without recursive symbols) are patterns of sort Map.

Definition 2.53. The *standard map model* is an $(S^{\mathsf{Map}}, \Sigma^{\mathsf{Map}})$-model (where the standard interpretation for the definiteness symbols are omitted)

$$M^{\mathsf{Map}} = (\{M^{\mathsf{Map}}_{\mathsf{Nat}}, M^{\mathsf{Map}}_{\mathsf{Map}}\}, \{\mathsf{nil}_{M^{\mathsf{Map}}}, \mathsf{emp}_{M^{\mathsf{Map}}}, (\_{\mapsto}\_)_{M^{\mathsf{Map}}}, (\_{*}\_)_{M^{\mathsf{Map}}}\})$$

where

1. $M^{\mathsf{Map}}_{\mathsf{Nat}} = \mathbb{N}$ and $M^{\mathsf{Map}}_{\mathsf{Map}} = \mathbb{H}$, defined in Section 2.6;

2. $\mathsf{nil}_{M^{\mathsf{Map}}} = \{0\}$;

3. $\mathsf{emp}_{M^{\mathsf{Map}}} = \{\emptyset\}$, where $\emptyset \in \mathbb{H}$ denotes the empty heap;

4. $(\_\mapsto\_)_{M^{\mathsf{Map}}}(m, n) = \{h_{m,n}\}$ for every $m, n \in \mathbb{N}$ with $m \neq 0$, where $h_{m,n}$ is the partial function that maps m to n and is undefined anywhere else;

5. $(\_\mapsto\_)_{M^{\mathsf{Map}}}(0, n) = \emptyset$ for every $n \in \mathbb{N}$;

6. $(\_*\_)_{M^{\mathsf{Map}}}(h_1, h_2) = h_1 \mathbin{\dot\cup} h_2$ for every $h_1, h_2 \in \mathbb{H}$ that are disjoint;

7. $(\_*\_)_{M^{\mathsf{Map}}}(h_1, h_2) = \emptyset$ for every $h_1, h_2 \in \mathbb{H}$ that are not disjoint.

Proposition 2.18 ([2, Proposition 9.2])**.** *$M^{\mathsf{Map}} \vDash \Gamma^{\mathsf{Map}}$. In addition, $\vDash_{\mathsf{SL}} \varphi$ iff $M^{\mathsf{Map}} \vDash \varphi$ for any SL formula φ.*

In other words, SL can be regarded as an instance/fragment of matching logic when we fix the underlying model to be M^{Map}.

2.13.3 Matching logic proof system $\mathcal{P}$

Matching logic has a conditional sound and complete Hilbert-style proof system $\mathcal{P}$, as shown in Figure 2.11. We use $\Gamma \vdash_{\mathcal{P}} \varphi$ to denote the corresponding provability relation. We call $\mathcal{P}$ a conditional proof system because it requires the definedness symbols and axioms in Definition 2.49. There are proof rules of $\mathcal{P}$ that use equality "=" and membership "$\in$", both requiring the definedness symbols and axioms. Therefore, $\mathcal{P}$ cannot be used on theories that do not have the definedness symbols or axioms. The completeness of $\mathcal{P}$ is proved by a reduction from matching logic to pure predicate logic with equality, which is FOL extended with a built-in equality symbol that has no function symbols. Through the reduction, the completeness of matching logic is reduced to the completeness of pure predicate logic with equality. The definedness symbols and axioms are needed for defining equality and membership, which are needed for mimicking the proofs of pure predicate logic with equality in matching logic.

Definition 2.54. For a matching logic symbol $\sigma \in \Sigma$, we write $C_\sigma[\varphi]$ to mean a pattern of the form $\sigma(\psi_1, \ldots, \psi_{i-1}, \varphi, \psi_{i+1}, \ldots, \psi_n)$.

Theorem 2.19 ([2, Theorem 11.2])**.** For any matching logic theory Γ that includes the definedness symbols and axioms in Definition 2.49, $\Gamma \vDash \varphi$ iff $\Gamma \vdash_{\mathcal{P}} \varphi$, for any matching logic pattern φ.

(Propositional Tautology)	φ, if φ is a proposition tautology
(Modus Ponens)	$\dfrac{\varphi_1 \quad \varphi_1 \to \varphi_2}{\varphi_2}$
(Functional Substitution)	$(\forall x \,.\, \varphi) \wedge (\exists y \,.\, \varphi' = y) \to \varphi[\varphi'/x]$ if $y \notin \mathit{freeVar}(\varphi')$
$(\forall)$	$\forall x \,.\, (\varphi_1 \to \varphi_2) \to (\varphi_1 \to \forall x \,.\, \varphi_2)$ if $x \notin \mathit{freeVar}(\varphi_1)$
(Universal Generalization)	$\dfrac{\varphi}{\forall x \,.\, \varphi}$
(Equality Introduction)	$\varphi = \varphi$
(Equality Elimination)	$(\varphi_1 = \varphi_2) \wedge \psi[\varphi_1/x] \to \psi[\varphi_2/x]$
(Membership Introduction)	$\dfrac{\varphi}{\forall x \,.\, (x \in \varphi)}$ if $x \notin \mathit{freeVar}(\varphi)$
(Membership Elimination)	$\dfrac{\forall x \,.\, (x \in \varphi)}{\varphi}$ if $x \notin \mathit{freeVar}(\varphi)$
(Membership Variable)	$(x \in y) = (x = y)$
(Membership$_\neg$)	$(x \in \neg\varphi) = \neg(x \in \varphi)$
(Membership$_\wedge$)	$(x \in \varphi_1 \wedge \varphi_2) = (x \in \varphi_1) \wedge (x \in \varphi_2)$
(Membership$_\exists$)	$(x \in \exists y \,.\, \varphi) = \exists y \,.\, (x \in \varphi)$, if x and y are distinct.
(Membership Symbol)	$x \in C_\sigma[\varphi] = \exists y \,.\, (y \in \varphi) \wedge (x \in C_\sigma[y])$ if $y \notin \mathit{freeVar}(C_\sigma[\varphi])$

Figure 2.11: Conditional Sound and Complete Proof System $\mathcal{P}$ of Matching Logic

2.14 REACHABILITY LOGIC

Reachability logic [11], abbreviated as RL, is an approach to program verification using operational semantics. Unlike the other approaches such as Hoare-style verification, RL has a language-independent proof system that offers sound and relatively complete deduction for all programming languages. RL is the logic underlying the $\mathbb{K}$ framework (Section 2.15), which has been used to define the formal semantics of many large programming languages, from which their sound and relatively complete program verifiers are obtained using RL at no additional cost [3].

RL is parametric in a matching logic model for computation configurations. Specifically, fix a signature (of static program configurations) Σ^{Cfg} which may have various sorts and symbols, among which there is a distinguished sort Cfg. Fix a Σ^{Cfg}-model M^{Cfg} called the *configuration model*, where $M^{\mathsf{Cfg}}_{\mathsf{Cfg}}$ is the set of all computation configurations. RL formulas are called *reachability rules*, or simply *rules*, and have the form $\varphi_1 \Rightarrow \varphi_2$ where φ_1, φ_2 are matching logic Σ^{Cfg}-patterns. A *reachability system* S is a finite set of rules, which yields a transition system $T = (M^{\mathsf{Cfg}}_{\mathsf{Cfg}}, \xrightarrow{T})$ where $s \xrightarrow{T} t$ iff there exist $(\varphi_1 \Rightarrow \varphi_2) \in S$ and an M^{Cfg}-valuation ρ such that $s \in |\varphi_1|_{T,\rho}$ and $t \in |\varphi_2|_{T,\rho}$. A rule $\psi_1 \Rightarrow \psi_2$ is S*-valid*, written $S \vDash_{\mathsf{RL}} \psi_1 \Rightarrow \psi_2$, iff for all $M^{\mathsf{Cfg}}_{\mathsf{Cfg}}$-valuations ρ and $s \in |\psi_1|_{T,\rho}$, either there is an infinite trace $s \xrightarrow{T} t_1 \xrightarrow{T} t_2 \xrightarrow{T} \dots$ in T or there exists $t \in T$ such that $s\,(\xrightarrow{T})^*\,r$ and $t \in |\psi_2|_{T,\rho}$. Recall that $(\xrightarrow{T})^*$ is the reflexive and transitive closure of $\xrightarrow{T}$ in Section 2.1.

RL has a sound and relatively complete proof system, as shown in Figure 2.12. The proof system derives sequents of the form $A \vdash_C \varphi_1 \Rightarrow \varphi_2$, where A (called *axioms*) and C (called *circularities*) are finite sets of rules. The corresponding provability relation $S \vdash_{\mathsf{RL}} \psi_1 \Rightarrow \psi_2$ is given by $S \vdash_\emptyset \psi_1 \Rightarrow \psi_2$. In other words, we start with $A = S$ and $C = \emptyset$ in any RL proof. As the proof proceeds, more rules can be added to C via (Circularity) and then moved to A via (Transitivity), which can then be used via (Axiom). Note that (Consequence) consults the underlying configuration model M^{Cfg} for the semantic satisfaction relation, so the completeness of the RL proof system is relative to M^{Cfg}.

Theorem 2.20. Let S be a reachability system that satisfies the technical assumptions in [11]. For any $\psi_1 \Rightarrow \psi_2$, $S \vDash_{\mathsf{RL}} \psi_1 \Rightarrow \psi_2$ iff $S \vdash_{\mathsf{RL}} \psi_1 \Rightarrow \psi_2$.

2.15 $\mathbb{K}$ FRAMEWORK

$\mathbb{K}$ framework is an effort in realizing the ideal language framework vision in Figure 1.1. An easy way to understand $\mathbb{K}$ is to look at it as a meta-language that can define other programming languages. In Figure 2.13, we show an example $\mathbb{K}$ language definition of an

$$\text{(Axiom)} \qquad \frac{\varphi \Rightarrow \varphi' \in A}{A \vdash_C \varphi \Rightarrow \varphi'}$$

$$\text{(Reflexivity)} \qquad \frac{}{A \vdash_\emptyset \varphi \Rightarrow \varphi}$$

$$\text{(Transitivity)} \qquad \frac{A \vdash_C \varphi_1 \Rightarrow \varphi_2 \quad A \cup C \vdash \varphi_2 \Rightarrow \varphi_3}{A \vdash_C \varphi_1 \Rightarrow \varphi_3}$$

$$\text{(Logic Framing)} \qquad \frac{A \vdash_C \varphi \Rightarrow \varphi' \quad \psi \text{ is a FOL formula}}{A \vdash_C \varphi \wedge \psi \Rightarrow \varphi' \wedge \psi}$$

$$\text{(Consequence)} \qquad \frac{M^{\mathsf{Cfg}} \models \varphi_1 \to \varphi_1' \quad A \vdash_C \varphi_1' \Rightarrow \varphi_2' \quad M^{\mathsf{Cfg}} \models \varphi_2' \to \varphi_2}{A \vdash_C \varphi_1 \Rightarrow \varphi_2}$$

$$\text{(Case Analysis)} \qquad \frac{A \vdash_C \varphi_1 \Rightarrow \varphi \quad A \vdash_C \varphi_2 \Rightarrow \varphi}{A \vdash_C \varphi_1 \vee \varphi_2 \Rightarrow \varphi}$$

$$\text{(Abstraction)} \qquad \frac{A \vdash_C \varphi \Rightarrow \varphi' \quad X \cap \mathit{freeVar}(\varphi') = \emptyset}{A \vdash_C \exists X \,.\, \varphi \Rightarrow \varphi'}$$

$$\text{(Circularity)} \qquad \frac{A \vdash_{C \cup \{\varphi \Rightarrow \varphi'\}} \varphi \Rightarrow \varphi'}{A \vdash_C \varphi \Rightarrow \varphi'}$$

Figure 2.12: Sound and Relatively Complete Proof System of RL

imperative language IMP. In the 39-line definition, we completely define the formal syntax and the (executable) formal semantics of IMP, using a front-end language that is easy to understand. From this language definition, $\mathbb{K}$ can generate all language tools for IMP, including its parser, interpreter, verifier, etc.

We use IMP as an example to illustrate the main $\mathbb{K}$ features. There are two modules: `IMP-SYNTAX` defines the syntax and `IMP` defines the semantics using rewrite rules. Syntax is defined as BNF grammars. The keyword `syntax` leads production rules that can have attributes that specify the additional syntactic and/or semantic information. For example, in lines 11-12, we define the syntax of if-statements, as expected. A production rule can be associated with attributes, which are written in brackets. For example, the syntax of `if`-statements is defined in lines 11-12 and has the attribute `[strict(1)]`, meaning that the evaluation order is strict in the first argument, i.e., the condition of an `if`-statement. There are many other attributes. Some attributes (like `[strict(1)]`) have semantic meaning while the others are only used for parsing. For example, the attribute `[left]` in line 6 means that the binary construct "`+`" is left associative.

In the module `IMP`, we define the *configurations* of IMP and its formal semantics. A configuration (lines 23-25) is a constructor term that has all semantic information needed to

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

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

Figure 2.13: Complete Formal Semantics of IMP in 𝕂

execute programs. IMP configurations are simple, consisting of the IMP code and a program state that maps variables to values. We organize configurations using (semantic) cells: `</k>` is the cell of IMP code and `</state>` is the cell of program states. In the initial configuration (lines 24-25), `</state>` is empty and `</k>` contains the IMP program that we pass to 𝕂 for execution (represented by the special 𝕂 variable `$PGM`).

We define formal semantics using rewrite rules. In lines 26-27, we define the semantics of variable lookup, where we match on a variable `X` in the `</k>` cell and look up its value `I` in the `</state>` cell, by matching on the binding `X ↦ I`. Then, we rewrite `X` to `I`, denoted by `X ⇒ I` in the `</k>` cell in line 26. Rewrite rules in 𝕂 are similar to those in the rewrite engines such as Maude [32].

Chapter 3: TWO COMPLETENESS THEOREMS FOR MATCHING LOGIC

As we have seen in Section 2.13.3, the proof system $\mathcal{P}$ requires the definedness symbols and axioms. A natural question is: Is there a proof system of matching logic that does not require the definedness symbols or axioms and can be used to do formal reasoning in any theories?

We will present such a proof system, which we refer to as the proof system $\mathcal{H}$. Unlike $\mathcal{P}$, $\mathcal{H}$ does not require the definedness symbols or axioms so it can be used to do formal reasoning in any matching logic theories. We will prove the soundness of $\mathcal{H}$ (Theorem 3.1).

As for completeness of $\mathcal{H}$, we prove two important results. The first result is the definedness completeness of $\mathcal{H}$, stated in Theorem 3.8. It says that $\mathcal{H}$ is complete for any theory that includes the definedness symbols and axioms. Therefore, $\mathcal{H}$ is at least as good as $\mathcal{P}$. We prove the definedness completeness result by showing that all the proof rules of $\mathcal{P}$ are derivable using $\mathcal{H}$ and the definedness axioms.

The second completeness result is the local completeness of $\mathcal{H}$, stated in Theorem 3.13. For this result, we define two new relations $\Gamma \vDash^{loc} \varphi$ and $\Gamma \vdash^{loc}_{\mathcal{H}} \varphi$ (Definition 3.3). We call $\Gamma \vDash^{loc} \varphi$ the local validity relation and $\Gamma \vdash^{loc}_{\mathcal{H}} \varphi$ the local provability relation. These local relations are stronger than their (global) counterparts $\Gamma \vDash \varphi$ and $\Gamma \vdash_{\mathcal{H}} \varphi$, respectively, and when $\Gamma = \emptyset$, they are equivalent to their global counterparts. The local (soundness) and completeness result states that $\Gamma \vDash^{loc} \varphi$ iff $\Gamma \vdash^{loc}_{\mathcal{H}} \varphi$.

We summarize the soundness, the definedness completeness, and the local completeness of $\mathcal{H}$ in Figure 3.1. We know that the diagram commutes if $\Gamma = \emptyset$. We also know that the diagram does not commute for an arbitrary Γ. There exists a (nonempty) Γ and a pattern φ such that $\Gamma \vDash \varphi$ and $\Gamma \vdash_{\mathcal{H}} \varphi$, but $\Gamma \nvDash^{loc} \varphi$ and $\Gamma \nvdash^{loc}_{\mathcal{H}} \varphi$, and we give such a counterexample in Section 3.3. The local soundness and completeness of $\mathcal{H}$ shows that the two local relations are equivalent for all Γ and φ. However, we do not know whether the two global relations are also equivalent for all Γ and φ. More precisely, we do not know whether $\Gamma \vDash \varphi$ always implies $\Gamma \vdash_{\mathcal{H}} \varphi$; we call it the global completeness of $\mathcal{H}$. We only know that the implication holds when $\Gamma = \emptyset$, which is the local completeness result, and when Γ includes definedness, which is the definedness completeness result. Global completeness of $\mathcal{H}$ is still an open problem.

3.1 MATCHING LOGIC PROOF SYSTEM $\mathcal{H}$

We first need the following definition of application contexts.

$$\begin{tikzcd}
\substack{\Gamma \vdash_{\mathcal{H}} \varphi \\ \text{(Definition 3.2)}} \arrow[rr, Rightarrow, shift left, "\text{Soundness}"] & & \substack{\Gamma \vDash \varphi \\ \text{(Definition 2.48)}} \arrow[ll, Rightarrow, shift left, "\text{Definedness Completeness}"', "\text{if } \Gamma \text{ includes definedness}"] \\
& \text{diagram commutes if } \Gamma = \emptyset & \\
\substack{\Gamma \vdash^{loc}_{\mathcal{H}} \varphi \\ \text{(Definition 3.3)}} \arrow[uu, Rightarrow] \arrow[rr, Rightarrow, shift left, "\text{Local Soundness}"] & & \substack{\Gamma \vDash^{loc} \varphi \\ \text{(Definition 3.3)}} \arrow[uu, Rightarrow] \arrow[ll, Rightarrow, shift left, "\text{Local Completeness}"']
\end{tikzcd}$$

Figure 3.1: Known Relation among $\vDash$, $\vDash^{loc}$, $\vdash_{\mathcal{H}}$, and $\vdash^{loc}_{\mathcal{H}}$

Definition 3.1. Let C be a pattern and $\square$ be a distinguished variable that occurs exactly once in C. We call C an *application context* if $\square$ appears within a number of (nested) symbols. Formally, C is an application context if

1. C is $\square$; or

2. C is $C_\sigma[C']$ and C' is an application context. Note that $C_\sigma[C']$ is the shortcut of $\sigma(\varphi_1, \ldots, \varphi_{i-1}, C', \varphi_{i+1}, \ldots, \varphi_n)$ in Definition 2.54.

We write $C[\varphi]$ to mean $C[\varphi/\square]$.

The proof system $\mathcal{H}$ is shown in Figure 3.2. It has nine proof rules that can be divided to three categories. The first category consists of four proof rules: (Propositional Tautology), (Modus Ponens), ($\exists$-Quantifier), and ($\exists$-Generalization). These four proof rules belong to the complete axiomatization of pure predicate logic; see, e.g., [33]. The second category consists of three proof rules: (Propagation$_\vee$), (Propagation$_\exists$), and (Framing). These three proof rules characterize the behaviors of symbols and allow us to propagate logical reasoning through symbols. The third category contains two technical rules that are necessary for proving definedness completeness (Theorem 3.8) and local completeness (Theorem 3.14).

Definition 3.2. We use $\Gamma \vdash_{\mathcal{H}} \varphi$ to denote the provability relation defined by $\mathcal{H}$.

Note that all proof rules of $\mathcal{H}$ are general rules and do not depend on any special symbols such as the definedness symbols. Therefore, $\mathcal{H}$ can be used to do formal reasoning in any theories.

3.1.1 Soundness of $\mathcal{H}$

We will show that $\mathcal{H}$ is sound, that is, $\Gamma \vdash_{\mathcal{H}} \varphi$ implies $\Gamma \vDash \varphi$, stated in Theorem 3.1. We first prove Lemma 3.1, known as the substitution lemma.

(Propositional Tautology)	φ if φ is a propositional tautology over patterns
(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 $x \notin \mathit{freeVar}(\varphi_2)$
(Propagation$_\vee$)	$C_\sigma[\varphi_1 \vee \varphi_2] \to C_\sigma[\varphi_1] \vee C_\sigma[\varphi_2]$
(Propagation$_\exists$)	$C_\sigma[\exists x\,.\,\varphi] \to \exists x\,.\,C_\sigma[\varphi]$ if $x \notin \mathit{freeVar}(C_\sigma[\exists x\,.\,\varphi])$
(Framing)	$\dfrac{\varphi_1 \to \varphi_2}{C_\sigma[\varphi_1] \to C_\sigma[\varphi_2]}$
(Existence)	$\exists x\,.\,x$
(Singleton Variable)	$\neg(C_1[x \wedge \varphi] \wedge C_2[x \wedge \neg\varphi])$ C_1 and C_2 are application contexts

Figure 3.2: Sound and Complete Proof System $\mathcal{H}$ of Matching Logic

Lemma 3.1. For any M and M-valuation ρ, $|\varphi[y/x]|_{M,\rho} = |\varphi|_{M,\rho[\rho(y)/x]}$.

Proof. We do structural induction on φ.

If φ is z, distinct from x, we have $|z[y/x]|_{M,\rho} = |z|_{M,\rho} = \{\rho(z)\}$ and $|z|_{M,\rho[\rho(y)/x]} = \{\rho(z)\}$.

If φ is x, we have $|x[y/x]|_{M,\rho} = |y|_{M,\rho} = \{\rho(y)\}$ and $|x|_{M,\rho[\rho(y)/x]} = \{\rho(y)\}$.

If φ is $\sigma(\varphi_1, \ldots, \varphi_n)$, we have

$$\begin{aligned}
|\sigma(\varphi_1, \ldots, \varphi_n)[y/x]|_{M,\rho} &= |\sigma(\varphi_1[y/x], \ldots, \varphi_n[y/x])|_{M,\rho} \\
&= \sigma_M(|\varphi_1[y/x]|_{M,\rho}, \ldots, |\varphi_n[y/x]|_{M,\rho}) \\
&= \sigma_M(|\varphi_1|_{M,\rho[\rho(y)/x]}, \ldots, |\varphi_n|_{M,\rho[\rho(y)/x]}) \\
&= |\sigma(\varphi_1, \ldots, \varphi_n)|_{M,\rho[\rho(y)/x]}
\end{aligned}$$

If φ is $\varphi_1 \wedge \varphi_2$, we have $|(\varphi_1 \wedge \varphi_2)[y/x]|_{M,\rho} = |\varphi_1[y/x] \wedge \varphi_2[y/x]|_{M,\rho} = |\varphi_1[y/x]|_{M,\rho} \cap |\varphi_2[y/x]|_{M,\rho} = |\varphi_1|_{M,\rho[\rho(y)/x]} \cap |\varphi_2|_{M,\rho[\rho(y)/x]} = |\varphi_1 \wedge \varphi_2|_{M,\rho[\rho(y)/x]}$.

If φ is $\neg\varphi_1$, we have $|(\neg\varphi_1)[y/x]|_{M,\rho} = |\neg(\varphi_1[y/x])|_{M,\rho} = M \setminus |\varphi_1[y/x]|_{M,\rho} = M \setminus |\varphi_1|_{M,\rho[\rho(y)/x]} = |\neg\varphi_1|_{M,\rho[\rho(y)/x]}$.

If φ is $\exists z \,.\, \varphi_1$, we can assume that z is distinct from x or y by α-renaming. Then we have

$$\begin{aligned}
|(\exists z \,.\, \varphi_1)[y/x]|_{M,\rho} &= |\exists z \,.\, (\varphi_1[y/x])|_{M,\rho} \\
&= \bigcup_{a \in M} |\varphi_1[y/x]|_{M,\rho[a/z]} \\
&= \bigcup_{a \in M} |\varphi_1|_{M,[\rho[a/z](y)/x]} \\
&= \bigcup_{a \in M} |\varphi_1|_{M,\rho[a/z][\rho(y)/x]} \\
&= \bigcup_{a \in M} |\varphi_1|_{M,\rho[\rho(y)/x][a/z]} \\
&= |\exists z \,.\, \varphi_1|_{M,\rho[\rho(y)/x]}
\end{aligned}$$

Therefore, the conclusion holds by structural induction. QED.

Lemma 3.2. Let C be an application context. For any M and M-valuation ρ, we have

1. $|C[\bot]|_{M,\rho} = \emptyset$;
2. $|C[\varphi_1 \vee \varphi_2]|_{M,\rho} = |\varphi_1|_{M,\rho} \cup |\varphi_2|_{M,\rho}$;
3. $|C[\exists x \,.\, \varphi]|_{M,\rho} = \bigcup_{a \in M} |C[\varphi]|_{M,\rho[a/x]}$ if $x \notin \mathit{freeVar}(C[\exists x \,.\, \varphi])$;
4. $|\varphi_1|_{M,\rho} \subseteq |\varphi_2|_{M,\rho}$ implies $|C[\varphi_1]|_{M,\rho} \subseteq |C[\varphi_2]|_{M,\rho}$;
5. $|C[x \wedge \varphi]|_{M,\rho} \cap |C[x \wedge \neg\varphi]|_{M,\rho} = \emptyset$.

Proof. We do structural induction on C.

(Base Case). In this case, $C[\square]$ is $\square$ and $C[\varphi]$ is just φ. All propositions hold.

(Induction Step). Let us assume $C[\square] \equiv C_\sigma[C_1[\square]]$, where

$$C_\sigma[\square] \equiv \sigma(\psi_1, \ldots, \psi_{i-1}, \square, \psi_{i+1}, \ldots, \psi_n)$$

for some $\sigma \in \Sigma$ and C is an application context. By the inductive hypotheses, all propositions hold for C_1. For simplicity, let us define

$$\sigma_M^i(A) = \sigma_M(|\psi_1|_{M,\rho}, \ldots, |\psi_{i-1}|_{M,\rho}, A, |\psi_{i+1}|_{M,\rho}, \ldots, |\psi_n|_{M,\rho}))$$

for $A \subseteq M$. Note that σ_M^i is monotone, that is, $\sigma_M^i(A_1) \subseteq \sigma_M^i(A_2)$ if $A_1 \subseteq A_2$. Under the above notation, $|C_\sigma[\varphi]|_{M,\rho} = \sigma_M^i(|\varphi|_{M,\rho})$. We now prove (1)–(5).

For (1), we have $|C_\sigma[C_1[\bot]]|_{M,\rho} = \sigma_M^i(|C_1[\bot]|_{M,\rho}) = \sigma_M^i(\emptyset) = \emptyset$.

For (2), we have $|C_\sigma[C_1[\varphi_1 \vee \varphi_2]]|_{M,\rho} = \sigma^i_M(|C_1[\varphi_1 \vee \varphi_2]|_{M,\rho}) = \sigma^i_M(|C_1[\varphi_1]|_{M,\rho} \cup |C_1[\varphi_2]|_{M,\rho}) = \sigma^i_M(|C_1[\varphi_1]|_{M,\rho}) \cup \sigma^i_M(|C_1[\varphi_1]|_{M,\rho}) = |C_\sigma[C_1[\varphi_1]]|_{M,\rho} \cup |C_\sigma[C_1[\varphi_2]]|_{M,\rho}$.

For (3), we have $|C_\sigma[C_1[\exists x \,.\, \varphi]]|_{M,\rho} = \sigma^i_M(|C_1[\exists x \,.\, \varphi]|_{M,\rho}) = \sigma^i_M(\bigcup_a |C_1[\varphi]|_{M,\rho[a/x]})$. Because $x \notin \mathit{freeVar}(C_\sigma[C_1[\exists x \,.\, \varphi]])$, we have $\sigma^i_M(\bigcup_a |C_1[\varphi]|_{M,\rho[a/x]}) = \bigcup_a \sigma^i_M(|C_1[\varphi]|_{M,\rho[a/x]}) = \bigcup_a |C_\sigma[C_1[\varphi]]|_{M,\rho[a/x]}$.

For (4), we need to prove that $|C_\sigma[C_1[\varphi_1]]|_{M,\rho} \subseteq |C_\sigma[C_1[\varphi_2]]|_{M,\rho}$, that is, $\sigma^i_M(|C_1[\varphi_1]|_{M,\rho}) \subseteq \sigma^i_M(|C_1[\varphi_2]|_{M,\rho})$. Since σ^i_M is monotone, we only need to prove that $|C_1[\varphi_1]|_{M,\rho} \subseteq |C_1[\varphi_2]|_{M,\rho}$. The latter holds by the inductive hypotheses and $|\varphi_1|_{M,\rho} \subseteq |\varphi_2|_{M,\rho}$.

For (5), we do a case analysis. If $\rho(x) \in |\varphi|_{M,\rho}$, we have $|x \wedge \varphi|_{M,\rho} = \emptyset$, and thus $|C[x \wedge \varphi]|_{M,\rho} = \emptyset$. Otherwise, we have $|x \wedge \neg\varphi|_{M,\rho} = \emptyset$, and thus $|C[x \wedge \neg\varphi]|_{M,\rho} = \emptyset$.

Therefore, all propositions hold by structural induction. QED.

Lemma 3.3. For any model M, the following propositions hold:

1. $M \vDash \varphi$ for propositional tautology φ over patterns of the same sort;
2. $M \vDash \varphi_1$ and $M \vDash \varphi_1 \to \varphi_2$ imply $M \vDash \varphi_2$;
3. $M \vDash \varphi[y/x] \to \exists x \,.\, \varphi$;
4. $M \vDash \varphi_1 \to \varphi_2$ implies $M \vDash (\exists x \,.\, \varphi_1) \to \varphi_2$ if $x \notin \mathit{freeVar}(\varphi_2)$;
5. $M \vDash C_\sigma[\bot] \to \bot$;
6. $M \vDash C_\sigma[\varphi_1 \vee \varphi_2] \to C_\sigma[\varphi_1] \vee C_\sigma[\varphi_2]$;
7. $M \vDash C_\sigma[\exists x \,.\, \varphi] \to \exists x \,.\, C_\sigma[\varphi]$ if $x \notin \mathit{freeVar}(C_\sigma[\exists x \,.\, \varphi])$;
8. $M \vDash \varphi_1 \to \varphi_2$ implies $M \vDash C_\sigma[\varphi_1] \to C_\sigma[\varphi_2]$
9. $M \vDash \exists x \,.\, x$
10. $M \vDash \neg(C_1[x \wedge \varphi] \wedge C_2[x \wedge \neg\varphi])$

Proof. (1) and (2) are proved in [2, Proposition 2.8]. Note that $M \vDash \varphi_1 \to \varphi_2$ iff $|\varphi_1|_{M,\rho} \subseteq |\varphi_2|_{M,\rho}$ for all ρ (see [2, Proposition 2.6]). We will use this property to prove (3)–(8). In the following, let ρ be any valuation.

(3). By Lemma 3.1, $|\varphi[y/x]|_{M,\rho} = |\varphi|_{M,\rho[\rho(y)/x]} \subseteq \bigcup_a |\varphi|_{M,\rho[a/x]} = |\exists x \,.\, \varphi|_{M,\rho}$.

(4). We need to prove that $|\exists x \,.\, \varphi_1|_{M,\rho} \subseteq |\varphi_2|_{M,\rho}$, that is, $\bigcup_a |\varphi_1|_{M,\rho[a/x]} \subseteq |\varphi_2|_{M,\rho}$. We only need to prove that $|\varphi_1|_{M,\rho[a/x]} \subseteq |\varphi_2|_{M,\rho}$ for all $a \in M$. Because $x \notin \mathit{freeVar}(\varphi_1)$, we have $|\varphi_1|_{M,\rho[a/x]} = |\varphi_1|_{M,\rho}$. Thus, we only need to prove that $|\varphi_1|_{M,\rho} \subseteq |\varphi_2|_{M,\rho}$. The latter holds by assumption.

(5)–(8) and (10). These propositions are direct consequences of Lemma 3.2.

(9). We have $|\exists x \,.\, x|_{M,\rho} = \bigcup_a |x|_{M,\rho[a/x]} = \bigcup_a \{a\} = M$. QED.

We now state and prove the soundness of $\mathcal{H}$ in Theorem 3.1.

Theorem 3.1. $\mathcal{H}$ is sound, that is, $\Gamma \vdash_{\mathcal{H}} \varphi$ implies $\Gamma \vDash \varphi$.

Proof. The proof is standard. Because $\Gamma \vdash_{\mathcal{H}} \varphi$, there exists a Hilbert-style proof for φ under Γ. We do mathematical induction on the length of the Hilbert-style proof for φ under Γ.

(Base Case). In this case, $n = 1$. Therefore, φ is an axiom of $\mathcal{H}$ or $\varphi \in \Gamma$. If φ is an axiom of $\mathcal{H}$, we have $\Gamma \vDash \varphi$ by Lemma 3.3. If $\varphi \in \Gamma$, then we have $\Gamma \vDash \varphi$ by Definition 2.48.

(Induction Step). Suppose the proof length is $n + 1$ for some $n \geq 1$, as shown in the following:

$$\varphi_1, \ldots, \varphi_n, \varphi_{n+1} \quad \text{where } \varphi_{n+1} \equiv \varphi.$$

If φ_{n+1} is an axiom of $\mathcal{H}$ or $\varphi_{n+1} \in \Gamma$, we have $\Gamma \vDash \varphi_{n+1}$ as in the base case. If φ_{n+1} is the result of applying (Modus Ponens), ($\exists$-Generalization), or (Framing), we have $\Gamma \vDash \varphi$ by Lemma 3.3.

Therefore, we prove the soundness of $\mathcal{H}$ by induction. QED.

3.1.2 Important properties of $\mathcal{H}$

Firstly, note that the proof rules (Propositional Tautology), (Modus Ponens), ($\exists$-Quantifier), and ($\exists$-Generalization) form a complete axiomatization of (pure) predicate logic, which is FOL without function symbols. It leads us to Proposition 3.2.

Proposition 3.2. *Predicate logic reasoning is sound for matching logic.*

Throughout this thesis, we will say "by FOL reasoning" to mean that a certain reasoning step in matching logic can be accomplished by applying the four proof rules: (Propositional Tautology), (Modus Ponens), ($\exists$-Quantifier), and ($\exists$-Generalization).

Secondly, we prove that frame reasoning is sound for matching logic.

Proposition 3.3. *The following propositions hold:*

1. *If* $\Gamma \vdash_{\mathcal{H}} \varphi_i \to \varphi_i'$ *for* $1 \leq i \leq n$, *then* $\Gamma \vdash_{\mathcal{H}} \sigma(\varphi_1, \ldots, \varphi_n) \to \sigma(\varphi_1', \ldots, \varphi_n')$;

2. *If* $\Gamma \vdash_{\mathcal{H}} \varphi \to \varphi'$, *then* $\Gamma \vdash_{\mathcal{H}} C[\varphi] \to C[\varphi_i']$, *where* C *is an application context.*

Proof. To prove (1), we only need to prove all the following propositions:

$$\begin{gathered}
\Gamma \vdash_{\mathcal{H}} \sigma(\varphi_1, \varphi_2, \ldots, \varphi_{n-1}, \varphi_n) \to \sigma(\varphi'_1, \varphi_2, \ldots, \varphi_{n-1}, \varphi_n) \\
\Gamma \vdash_{\mathcal{H}} \sigma(\varphi'_1, \varphi_2, \ldots, \varphi_{n-1}, \varphi_n) \to \sigma(\varphi'_1, \varphi'_2, \ldots, \varphi_{n-1}, \varphi_n) \\
\cdots \\
\Gamma \vdash_{\mathcal{H}} \sigma(\varphi'_1, \varphi'_2, \ldots, \varphi'_{n-1}, \varphi_n) \to \sigma(\varphi'_1, \varphi'_2, \ldots, \varphi'_{n-1}, \varphi'_n)
\end{gathered}$$

These propositions can be directly proved by (Framing).

To prove (2), we do structural induction on C.

(Base Case). Suppose C is $\square$. In this case, the proposition holds.

(Induction Step). Suppose $C \equiv C_\sigma[C_1]$, where $\sigma \in \Sigma$ and C_1 is an application context. Then we have

$$\begin{aligned}
&\Gamma \vdash_{\mathcal{H}} \varphi \to \varphi' && \text{// assumption} \\
&\Gamma \vdash_{\mathcal{H}} C_1[\varphi] \to C_1[\varphi'] && \text{// inductive hypothesis} \\
&\Gamma \vdash_{\mathcal{H}} C_\sigma[C_1[\varphi]] \to C_\sigma[C_1[\varphi']] && \text{// (Framing)}
\end{aligned}$$

Therefore, (2) holds by structural induction. QED.

Thirdly, we show that certain logical reasoning can be propagated through application contexts. More specifically, logical reasoning that has a "disjunctive" semantics can be propagated through application contexts. This includes $\vee$ (disjunction) whose semantics is set union, $\exists$ (existential quantification) whose semantics is also set union, and $\bot$, which is the unit of disjunction.

Proposition 3.4. *Let C be an application context. The following propositions hold:*

1. $\Gamma \vdash_{\mathcal{H}} C[\bot] \leftrightarrow \bot$;
2. $\Gamma \vdash_{\mathcal{H}} C[\varphi_1 \vee \varphi_2] \leftrightarrow C[\varphi_1] \vee C[\varphi_2]$;
3. $\Gamma \vdash C[\exists x \,.\, \varphi] \leftrightarrow \exists x \,.\, C[\varphi]$, *if* $x \notin \mathit{freeVar}(C[\exists x \,.\, \varphi])$;
4. $\Gamma \vdash_{\mathcal{H}} C[\varphi_1 \vee \varphi_2]$ *iff* $\Gamma \vdash_{\mathcal{H}} C[\varphi_1] \vee C[\varphi_2]$;
5. $\Gamma \vdash C[\exists x \,.\, \varphi]$ *iff* $\Gamma \vdash_{\mathcal{H}} \exists x \,.\, C[\varphi]$, *if* $x \notin \mathit{freeVar}(C[\exists x \,.\, \varphi])$.

Proof. We do structural induction on C.

(Base Case). Suppose C is $\square$. In this case, all propositions hold.

(Induction Step). Suppose C is $C_\sigma[C_1]$ where C_1 is an application context. We prove (1)–(5) using the induction hypothesis about C_1.

(1,“$\to$”). By the inductive hypothesis, we have $\Gamma \vdash_{\mathcal{H}} C_1[\bot] \to \bot$. By (Framing), we have $\Gamma \vdash_{\mathcal{H}} C_\sigma[C_1[\bot]] \to C_\sigma[\bot]$, i.e., $\Gamma \vdash_{\mathcal{H}} C[\bot] \to C_\sigma[\bot]$. Therefore, we only need to prove that $\Gamma \vdash_{\mathcal{H}} C_\sigma[\bot] \to \bot$. Let x be any variable and ψ be any pattern.[1] We have $\Gamma \vdash_{\mathcal{H}} \bot \to (x \wedge \psi)$ and $\Gamma \vdash_{\mathcal{H}} \bot \to (x \wedge \neg\psi)$. By (Framing), we have $\Gamma \vdash_{\mathcal{H}} C_\sigma[\bot] \to C_\sigma[x \wedge \psi]$ and $\Gamma \vdash_{\mathcal{H}} C_\sigma[\bot] \to C_\sigma[x \wedge \neg\psi]$. Therefore, we have $\Gamma \vdash_{\mathcal{H}} C_\sigma[\bot] \to (C_\sigma[x \wedge \psi] \wedge C_\sigma[x \wedge \neg\psi])$. On the other hand, by (Singleton Variable), we have $\Gamma \vdash_{\mathcal{H}} \neg(C_\sigma[x \wedge \psi] \wedge C_\sigma[x \wedge \neg\psi])$. Therefore, $\Gamma \vdash_{\mathcal{H}} C_\sigma[\bot] \to \bot$.

(1,“$\leftarrow$”). By FOL reasoning.

(2,“$\to$”). Same as (1,“$\to$”) except that we use (Propagation$_\vee$).

(2,“$\leftarrow$”). We only need to prove $\Gamma \vdash_{\mathcal{H}} C[\varphi_i] \to C[\varphi_1 \vee \varphi_2]$ for $i \in \{1, 2\}$. They can be proved by applying frame reasoning (Proposition 3.3) on $\Gamma \vdash_{\mathcal{H}} \varphi_i \to \varphi_1 \vee \varphi_2$.

(3,“$\to$”). Same as (1,“$\to$”) except that we use (Propagation$_\exists$).

(3,“$\leftarrow$”). We only need to prove $\Gamma \vdash_{\mathcal{H}} (\exists x \,.\, C[\varphi]) \to C[\exists x \,.\, \varphi]$. By ($\exists$-Generalization), we only need to prove $\Gamma \vdash_{\mathcal{H}} C[\varphi] \to C[\exists x \,.\, \varphi]$. It can be proved by applying frame reasoning (Proposition 3.3) on $\Gamma \vdash_{\mathcal{H}} \varphi \to \exists x \,.\, \varphi$.

(4) and (5) are direct consequences of (1)–(3).

Therefore, the propositions hold by structural induction. QED.

Lemma 3.4. For an application context C, $\Gamma \vdash_{\mathcal{H}} \varphi$ implies $\Gamma \vdash_{\mathcal{H}} \neg C[\neg\varphi]$.

Proof.

1	φ	hypothesis
2	$\neg\varphi \to \bot$	by 1, FOL reasoning
3	$C[\neg\varphi] \to C[\bot]$	by 2, (Framing)
4	$C[\bot] \to \bot$	by (Propagation)
5	$C[\neg\varphi] \to \bot$	by 3 and 4, FOL reasoning
6	$\neg C[\neg\varphi]$	by 5, FOL reasoning

QED.

Finally, we show that logical equivalence propagates through any context, as expected. A *context* C (not just an application context) is a pattern with a distinguished variable $\square$. We use $C[\varphi]$ to denote the result of in-place replacing $\square$ with φ.

Proposition 3.5. *For any context C (not just an application context), $\Gamma \vdash_{\mathcal{H}} \varphi_1 \leftrightarrow \varphi_2$ implies $\Gamma \vdash_{\mathcal{H}} C[\varphi_1] \leftrightarrow C[\varphi_2]$.*

[1] The proof of $\Gamma \vdash_{\mathcal{H}} C_\sigma[\bot] \to \bot$ presented here is credited to Mircea Sebe.

Proof. We do structural induction on C. If C is $\square$, the conclusion holds. If C has one of the following forms: $\neg C'$, $\psi \wedge C'$, $C' \wedge \psi$, or $\exists x \,.\, C'$, where C' is a context, the conclusion holds by FOL reasoning. If C has the form $C_\sigma[C']$, the conclusion holds by Proposition 3.3. Therefore, the conclusion holds by structural induction. QED.

Proposition 3.5 allows us to replace any two logically equivalent patterns in any context.

3.1.3 Relation to modal logic proof rules

There is a close relation between matching logic and modal logic. More specifically, matching logic symbols and modal operators are dual to each other.

Theorem 3.6. Given a matching logic symbol σ, we define its dual as $\sigma^d(\varphi_1, \ldots, \varphi_n) \equiv \neg\sigma(\neg\varphi_1, \ldots, \neg\varphi_n)$. Then we have:

- (K): $\emptyset \vdash_{\mathcal{H}} \sigma^d(\varphi_1 \to \varphi_1', \ldots, \varphi_n \to \varphi_n') \to (\sigma^d(\varphi_1, \ldots, \varphi_n) \to \sigma^d(\varphi_1', \ldots, \varphi_n'))$;
- (N): $\emptyset \vdash_{\mathcal{H}} \varphi_i$ implies $\emptyset \vdash_{\mathcal{H}} \sigma^d(\varphi_1, \ldots, \varphi_i, \ldots, \varphi_n)$.
- (Barcan): $\emptyset \vdash_{\mathcal{H}} (\forall x \,.\, \sigma^d(\ldots, \varphi_i, \ldots)) \to \sigma^d(\ldots, \forall x \,.\, \varphi_i, \ldots)$ if x does not occur free in the "$\ldots$" part.

Proof. Let $C_\sigma[\square] = \sigma(\varphi_1, \ldots, \varphi_{i-1}, \square, \varphi_{i+1}, \ldots, \varphi_n)$.

(K). By FOL reasoning, we only need to prove the case of one argument, that is, $\emptyset \vdash_{\mathcal{H}} \neg C_\sigma[\neg(\varphi \to \varphi')] \to (\neg C_\sigma[\neg\varphi] \to \neg C_\sigma[\neg\varphi'])$. By FOL reasoning, we only need to prove $\emptyset \vdash_{\mathcal{H}} C_\sigma[\varphi \wedge \varphi'] \vee C_\sigma[\neg\varphi] \vee \neg C_\sigma[\neg\varphi']$. By Proposition 3.4, we need to prove $\emptyset \vdash_{\mathcal{H}} C_\sigma[(\varphi \wedge \varphi') \vee \neg\varphi] \vee \neg C_\sigma[\neg\varphi']$, i.e., $\emptyset \vdash_{\mathcal{H}} C_\sigma[\varphi' \vee \neg\varphi] \vee \neg C_\sigma[\neg\varphi']$. By Proposition 3.4, we need to prove $\vdash_{\mathcal{H}} C_\sigma[\varphi'] \vee C_\sigma[\neg\varphi] \vee \neg C_\sigma[\neg\varphi']$. The latter holds by FOL reasoning.

(N). It is a direct consequence of Lemma 3.4, where we let C to be C_σ.

(Barcan). By unfolding $\forall x$ to $\neg\exists x \neg$, we need to prove that $\emptyset \vdash_{\mathcal{H}} (\neg\exists x \,.\, C_\sigma[\neg\varphi_i]) \to \neg C_\sigma[\exists x \,.\, \neg\varphi_i]$. Therefore, we need to prove that $\emptyset \vdash_{\mathcal{H}} C_\sigma[\exists x \,.\, \neg\varphi_i] \to \exists x \,.\, C_\sigma[\neg\varphi_i]$. The latter is provable by (Propagation$_\exists$). QED.

These above proof rules are also proof rules of polyadic modal logic and hyrbid logic [34, 35]. If we let $n = 1$, we obtain the standard (K) rule and (N) rule of modal logic K (Figure 2.3).

We will prove that the proof system $\mathcal{H}$ is complete for every theory that contains the following definedness symbols and axioms in Definition 2.49:

$$\begin{array}{ll} \lceil \_ \rceil_s^{s'} \in \Sigma_{s,s'} & \text{// definedness symbols} \\ \lceil x : s \rceil & \text{// (Definedness) axioms} \end{array}$$

This result is called definedness completeness, stated in Theorem 3.8. In other words, $\mathcal{H}$ is as good as the conditional sound and complete proof system $\mathcal{P}$ in Section 2.13.3, but unlike $\mathcal{P}$, it does not rely on the existence of definedness symbols or axioms and can be used to do formal reasoning with any theory. In fact, we will prove definedness completeness by showing that all the proof rules of $\mathcal{P}$ are derivable using $\mathcal{H}$ and the definedness axioms, that is:

$$\lceil x : s \rceil \vdash_{\mathcal{H}} \text{(all the proof rules of } \mathcal{P}\text{)}$$

Throughout this section we will assume that Γ is a theory that includes the definedness axioms. To simplify our notation we feel free to drop the sorts when they are not important.

Let us first go through all the proof rules of $\mathcal{P}$ and see which of them are already known to be derivable using $\mathcal{H}$. The proof system $\mathcal{P}$ has 14 proof rules in total (Figure 2.11). (Propositional Tautology) and (Modus Ponens) are also proof rules of $\mathcal{H}$ so they are derivable. ($\forall$) and (Universal Generalization) are derivable by FOL reasoning. Therefore, we only need to consider the (Functional Substitution) rule, two (Equality) rules, and seven (Membership) rules.

Lemma 3.5. $\Gamma \vdash_{\mathcal{H}} \varphi_1 \leftrightarrow \varphi_2$ implies $\Gamma \vdash_{\mathcal{H}} \varphi_1 = \varphi_2$.

Proof.

$$\begin{array}{r|ll} 1 & \varphi_1 \leftrightarrow \varphi_2 & \text{hypothesis} \\ 2 & \neg\lceil\neg(\varphi_1 \leftrightarrow \varphi_2)\rceil & \text{by 1, Lemma 3.4} \\ 3 & \varphi_1 = \varphi_2 & \text{by 2, definition of equality} \end{array}$$

QED.

Lemma 3.6. (Equality Introduction) can be proved in $\mathcal{H}$.

Proof.

$$\begin{array}{r|ll} 1 & \varphi \leftrightarrow \varphi & \text{propositional tautology} \\ 2 & \varphi = \varphi & \text{by 1, Lemma 3.5} \end{array}$$

QED.

Lemma 3.7. (Membership Introduction) can be proved in $\mathcal{H}$.

Proof.

1	φ	hypothesis
2	$\varphi \to (x \to \varphi)$	(Propositional Tautology)
3	$x \to \varphi$	by 1 and 2, (Modus Ponens)
4	$x \to x$	(Propositional Tautology)
5	$x \to x \land \varphi$	by 3 and 4, FOL reasoning
6	$\lceil x \rceil \to \lceil x \land \varphi \rceil$	by 5, (Framing)
7	$\lceil x \rceil$	definedness axiom
8	$\lceil x \land \varphi \rceil$	by 6 and 7, (Modus Ponens)
9	$x \in \varphi$	by 8, definition of membership
10	$\forall x \,.\, (x \in \varphi)$	by 9, FOL reasoning

QED.

Lemma 3.8. (Membership Elimination) can be proved in $\mathcal{H}$.

Proof.

1	$\forall x \,.\, (x \in \varphi)$	hypothesis
2	$(\forall x \,.\, (x \in \varphi)) \to x \in \varphi$	FOL reasoning
3	$x \in \varphi$	by 1 and 2, (Modus Ponens)
4	$\lceil x \land \varphi \rceil$	by 3, definition of membership
5	$\neg(\lceil x \land \varphi \rceil \land (x \land \neg\varphi))$	(Singleton Variable)
6	$\lceil x \land \varphi \rceil \to (x \to \varphi)$	by 5, FOL reasoning
7	$x \to \varphi$	by 4 and 6, (Modus Ponens)
8	$\forall x \,.\, (x \to \varphi)$	by 7, FOL reasoning
9	$(\exists x \,.\, x) \to \varphi$	by 8, FOL reasoning
10	$\exists x \,.\, x$	(Existence)
11	φ	by 10 and 9, (Modus Ponens)

QED.

Lemma 3.9. (Membership Variable) can be proved in $\mathcal{H}$.

Proof. By Lemma 3.5, we to prove $\vdash_{\mathcal{H}} (x \in y) \to (x = y)$ and $\vdash_{\mathcal{H}} (x = y) \to (x \in y)$. We first prove $\vdash_{\mathcal{H}} (x = y) \to (x \in y)$.

1	$\lceil x \rceil$	definedness axiom
2	$\lceil x \rceil \vee \lceil y \rceil$	by 1, FOL reasoning
3	$\lceil x \vee y \rceil$	by 2, Proposition 3.4
4	$\lceil \neg(x \leftrightarrow y) \vee (x \wedge y) \rceil$	by 3, FOL reasoning
5	$\lceil \neg(x \leftrightarrow y) \rceil \vee \lceil x \wedge y \rceil$	by 4, Proposition 3.4
6	$\neg\lceil \neg(x \leftrightarrow y) \rceil \to \lceil x \wedge y \rceil$	by 5, FOL reasoning
7	$(x = y) \to (x \in y)$	by 6, definition

We then prove $\vdash_{\mathcal{H}} (x \in y) \to (x = y)$.

1	$\neg(\lceil x \wedge y \rceil \wedge \lceil x \wedge \neg y \rceil)$	by (Singleton Variable)
2	$\neg(\lceil x \wedge y \rceil \wedge \lceil \neg x \wedge y \rceil)$	by (Singleton Variable)
3	$\lceil x \wedge y \rceil \to \neg\lceil x \wedge \neg y \rceil$	by 1, FOL reasoning
4	$\lceil x \wedge y \rceil \to \neg\lceil \neg x \wedge y \rceil$	by 2, FOL reasoning
5	$\lceil x \wedge y \rceil \to \neg\lceil x \wedge \neg y \rceil \wedge \neg\lceil \neg x \wedge y \rceil$	by 3 and 4, FOL reasoning
6	$\lceil x \wedge y \rceil \to \neg(\lceil x \wedge \neg y \rceil \vee \lceil \neg x \wedge y \rceil)$	by 5, FOL reasoning
7	$\lceil x \wedge y \rceil \to \neg\lceil (x \wedge \neg y) \vee (\neg x \wedge y) \rceil$	by 6, Proposition 3.4
8	$\lceil x \wedge y \rceil \to \neg\lceil \neg(x \leftrightarrow y) \rceil$	by 7, FOL reasoning
9	$(x \in y) \to (x = y)$	by 8, definition

QED.

Lemma 3.10. (Membership$_\neg$) can be proved in $\mathcal{H}$.

Proof. We first prove $\vdash_{\mathcal{H}} (x \in \neg\varphi) \to \neg(x \in \varphi)$.

1	$\neg(\lceil x \wedge \varphi \rceil \wedge \lceil x \wedge \neg\varphi \rceil)$	by (Singleton Variable)
2	$\lceil x \wedge \neg\varphi \rceil \to \neg\lceil x \wedge \varphi \rceil$	by 1, FOL reasoning
3	$(x \in \neg\varphi) \to \neg(x \in \varphi)$	by 2, definition

We then prove $\vdash_{\mathcal{H}} \neg(x \in \varphi) \to (x \in \neg\varphi)$.

1	$\lceil x \rceil$	definedness axiom
2	$\lceil (x \wedge \varphi) \vee (x \wedge \neg\varphi) \rceil$	by 1, FOL reasoning
3	$\lceil x \wedge \varphi \rceil \vee \lceil x \wedge \neg\varphi \rceil$	by 2, Proposition 3.4
4	$\neg\lceil x \wedge \varphi \rceil \to \lceil x \wedge \neg\varphi \rceil$	by 3, FOL reasoning
5	$\neg(x \in \varphi) \to (x \in \neg\varphi)$	by 4, definition

QED.

Lemma 3.11. $\vdash_{\mathcal{H}} (x \in (\varphi_1 \vee \varphi_2)) \leftrightarrow (x \in \varphi_1) \vee (x \in \varphi_2)$.

Proof. Use (Propagation$_\vee$) and FOL reasoning. QED.

Lemma 3.12. ($\textsc{Membership}_\wedge$) can be proved in $\mathcal{H}$.

Proof. Use Lemma 3.10 and 3.11, and the fact that $\vdash_{\mathcal{H}} \varphi_1 \wedge \varphi_2 \leftrightarrow \neg(\neg\varphi_1 \vee \neg\varphi_2)$. QED.

Lemma 3.13. ($\textsc{Membership}_\exists$) can be proved in $\mathcal{H}$.

Proof. Use ($\textsc{Propagation}_\exists$) and FOL reasoning. QED.

The following is a useful lemma about definedness symbols.

Lemma 3.14. $\vdash_{\mathcal{H}} C[\varphi] \to \lceil\varphi\rceil$ for any application context C.

Proof. Let x be a fresh variable in the following proof.

1	$\lceil x \rceil$	definedness axiom
2	$\lceil x \rceil \vee \lceil \varphi \rceil$	by 1, FOL reasoning
3	$\lceil x \vee \varphi \rceil$	by 2, Proposition 3.4
4	$\lceil x \wedge \neg\varphi \vee \varphi \rceil$	by 3, FOL reasoning
5	$\lceil x \wedge \neg\varphi \rceil \vee \lceil \varphi \rceil$	by 4, Proposition 3.4
6	$C[x \wedge \varphi] \to \neg\lceil x \wedge \neg\varphi \rceil$	by ($\textsc{Singleton Variable}$)
7	$\neg\lceil x \wedge \neg\varphi \rceil \to \lceil \varphi \rceil$	by 5, FOL reasoning
8	$C[x \wedge \varphi] \to \lceil \varphi \rceil$	by 6 and 7, FOL reasoning
9	$\forall x \,.\, (C[x \wedge \varphi] \to \lceil \varphi \rceil)$	by 8, FOL reasoning
10	$(\exists x \,.\, C[x \wedge \varphi]) \to \lceil \varphi \rceil$	by 9, FOL reasoning
11	$\varphi \to (\exists x \,.\, x) \wedge \varphi$	by ($\textsc{Existence}$)
12	$\varphi \to \exists x \,.\, (x \wedge \varphi)$	by 11, FOL reasoning
13	$C[\varphi] \to C[\exists x \,.\, (x \wedge \varphi)]$	by 12, ($\textsc{Framing}$)
14	$C[\exists x \,.\, (x \wedge \varphi)] \to \lceil \varphi \rceil$	by 10, Proposition 3.4
15	$C[\varphi] \to \lceil \varphi \rceil$	by 13, 14, FOL reasoning

QED.

Corollary 3.1. $\vdash_{\mathcal{H}} C_\sigma[\varphi] \to \lceil\varphi\rceil$ and $\vdash_{\mathcal{H}} \lfloor\varphi\rfloor \to \neg C_\sigma[\neg\varphi]$ for every symbol σ. Also, $\vdash_{\mathcal{H}} \varphi \to \lceil\varphi\rceil$ and $\vdash_{\mathcal{H}} \lfloor\varphi\rfloor \to \varphi$.

Proof. Let C be C_σ and $\square$ in Lemma 3.14, respectively. QED.

We state and prove a deduction theorem of $\mathcal{H}$.

Theorem 3.7. Let Γ be a theory that contains definedness. For any φ and ψ, if $\Gamma \cup \{\psi\} \vdash_{\mathcal{H}} \varphi$ and the proof does not use ($\exists$-$\textsc{Generalization}$) on any free variables of ψ, then $\Gamma \vdash_{\mathcal{H}} \lfloor\psi\rfloor \to \varphi$. In particular, if ψ is closed, then $\Gamma \cup \{\psi\} \vdash_{\mathcal{H}} \varphi$ implies $\Gamma \vdash_{\mathcal{H}} \lfloor\psi\rfloor \to \varphi$.

The condition regarding ($\exists$-Generalization) is standard. For example, the deduction theorem for FOL also has a similar condition [36]. The proof of Theorem 3.7 is standard, by using mathematical induction on the length of the proof of $\Gamma \cup \{\psi\} \vdash_{\mathcal{H}} \varphi$. In the following, we give a semantic argument and explain why the axiom ψ becomes $\lfloor \psi \rfloor$ when we move it from the left-hand side of $\vdash_{\mathcal{H}}$ to the right-hand side.

Suppose $\Gamma \cup \{\psi\} \vDash \varphi$ where ψ is closed. By definition, $M \vDash \Gamma$ and $M \vDash \psi$ implies $M \vDash \varphi$ for every M. In other words, if $M \vDash \Gamma$, then we have

$$\text{“}\psi \text{ holds in } M\text{”} \quad \text{implies} \quad \text{“}\varphi \text{ holds in } M\text{”}$$

The above implication can be equivalently stated as $M \vDash \lfloor \psi \rfloor \to \varphi$, because if ψ does not hold in M, $\lfloor \psi \rfloor$ is equivalent to $\bot$, and the implication holds. Otherwise, $\lfloor \psi \rfloor$ is equivalent to $\top$, and the implication holds iff φ holds. Therefore, an equivalent statement of $\Gamma \cup \{\psi\} \vDash \varphi$ is that for every M, if $M \vDash \Gamma$ then $M \vDash \lfloor \psi \rfloor \to \varphi$. The latter is equivalent to $\Gamma \vDash \lfloor \psi \rfloor \to \varphi$ by definition.

Note that $\Gamma \vDash \psi \to \varphi$ is too strong as a conclusion, for it requires that ψ is always included by φ, even in models where ψ does not hold. Here is a simple counterexample: $\Gamma \cup \{\psi\} \vDash \lfloor \psi \rfloor$ does not imply $\Gamma \vDash \psi \to \lfloor \psi \rfloor$. To better understand it, let us compare the following three statements: (a) $\Gamma \cup \{\psi\} \vDash \varphi$; (b) $\Gamma \vDash \lfloor \psi \rfloor \to \varphi$; and (c) $\Gamma \vDash \psi \to \varphi$, where we assume that ψ is closed for simplicity. By definition, (a) means that for all models M such that $M \vDash \Gamma$ and $M \vDash \psi$, we have $M \vDash \varphi$. Here, $M \vDash \psi$ means that $|\psi|_{M,\rho} = M$ for all ρ. Statement (b) means that for all models M such that $M \vDash \Gamma$, we have $M \vDash \lfloor \psi \rfloor \to \varphi$. Compared with (a), (b) checks more models. It checks not only models where ψ holds but also those where ψ does not hold. For models M where ψ hold, we can easily conclude that $M \vDash \lfloor \psi \rfloor \to \varphi$ because by (a), we have the stronger result $M \vDash \varphi$. For those models M where ψ does not hold, we have that $|\psi|_{M,\rho} \neq M$ for all ρ. This means that $|\lfloor \psi \rfloor|_{M,\rho} = \emptyset$ for all ρ, and thus $|\lfloor \psi \rfloor \to \varphi|_{M,\rho} = M$ no matter what φ is. This way, (a) implies (b), even if (b) checks more models than (a). The above reasoning fails for (c) because we cannot conclude anything on models where ψ does not hold.

Next, we prove Theorem 3.7.

Proof of Theorem 3.7. We do mathematical induction on the length of the proof $\Gamma \cup \{\psi\} \vdash_{\mathcal{H}} \varphi$.

(Base Case). Suppose the proof length is 1. In this case, φ is an axiom of $\mathcal{H}$ or $\varphi \in \Gamma \cup \{\psi\}$. We have $\Gamma \vdash_{\mathcal{H}} \lfloor \psi \rfloor \to \varphi$ (noticing Corollary 3.1 if φ is ψ).

(Induction Step). Suppose the proof $\Gamma \cup \{\psi\} \vdash_{\mathcal{H}} \varphi$ has $n+1$ steps:

$$\varphi_1, \ldots, \varphi_n, \varphi.$$

We now do a case analysis on how φ is proved.

Suppose φ is an axiom in $\mathcal{H}$ or $\varphi \in \Gamma \cup \{\psi\}$. We have $\Gamma \vdash_{\mathcal{H}} \lfloor\psi\rfloor \to \varphi$ for the same reason as (Base Case).

Suppose φ is proved by applying (Modus Ponens) on φ_i and φ_j for some $1 \le i, j \le n$, where φ_j has the form $\varphi_i \to \varphi$. By the induction hypotheses, $\Gamma \vdash_{\mathcal{H}} \lfloor\psi\rfloor \to \varphi_i$ and $\Gamma \vdash_{\mathcal{H}} \lfloor\psi\rfloor \to (\varphi_i \to \varphi)$. By FOL reasoning, $\Gamma \vdash_{\mathcal{H}} \lfloor\psi\rfloor \to \varphi$.

Suppose φ is proved by applying ($\exists$-Generalization) on $\varphi_1 \to \varphi_2$, and φ has the form $(\exists x\,.\,\varphi_1) \to \varphi_2$, where $x \notin \mathit{freeVar}(\varphi_2)$. By the induction hypothesis, $\Gamma \vdash_{\mathcal{H}} \lfloor\psi\rfloor \to (\varphi_1 \to \varphi_2)$. Therefore, $\Gamma \vdash_{\mathcal{H}} \varphi_1 \to (\lfloor\psi\rfloor \to \varphi_2)$. By assumption, the proof of φ does not apply ($\exists$-Generalization) on any free variable of ψ. Therefore, $x \notin \mathit{freeVar}(\psi)$, and we have $\Gamma \vdash_{\mathcal{H}} (\exists x\,.\,\varphi_1) \to (\lfloor\psi\rfloor \to \varphi_2)$ by ($\exists$-Generalization). Finally, we have $\Gamma \vdash_{\mathcal{H}} \lfloor\psi\rfloor \to ((\exists x\,.\,\varphi_1) \to \varphi_2)$ by FOL reasoning.

Suppose φ is proved by applying (Framing) on φ_i for some $1 \le i \le n$, then φ_i must have the form $\varphi_i' \to \varphi_i''$, and φ must have the form $C_\sigma[\varphi_i'] \to C_\sigma[\varphi_i'']$ for some σ. By the induction hypothesis, $\Gamma \vdash_{\mathcal{H}} \lfloor\psi\rfloor \to (\varphi_i' \to \varphi_i'')$. We can prove $\Gamma \vdash_{\mathcal{H}} \lfloor\psi\rfloor \to (C_\sigma[\varphi_i'] \to C_\sigma[\varphi_i''])$ as follows:

1	$\lfloor\psi\rfloor \to (\varphi_i' \to \varphi_i'')$	hypothesis
2	$\varphi_i' \to \varphi_i'' \vee \lceil\neg\psi\rceil$	by 1, FOL reasoning
3	$C_\sigma[\lceil\neg\psi\rceil] \to \lceil\neg\psi\rceil$	Corollary 3.1
4	$C_\sigma[\varphi_i'] \to C_\sigma[\varphi_i'' \vee \lceil\neg\psi\rceil]$	by 2, (Framing)
5	$C_\sigma[\varphi_i'] \to C_\sigma[\varphi_i''] \vee C_\sigma[\lceil\neg\psi\rceil]$	by 4, Proposition 3.4
6	$C_\sigma[\varphi_i''] \vee C_\sigma[\lceil\neg\psi\rceil] \to C_\sigma[\varphi_i''] \vee \lceil\neg\psi\rceil$	by 3, FOL reasoning
7	$C_\sigma[\varphi_i'] \to C_\sigma[\varphi_i''] \vee \lceil\neg\psi\rceil$	by 5, 6, FOL reasoning
8	$\lfloor\psi\rfloor \to (C_\sigma[\varphi_i'] \to C_\sigma[\varphi_i''])$	by 7, FOL reasoning

Therefore, the conclusion holds by induction. QED.

Next, we continue to prove the proof rules of $\mathcal{P}$.

Lemma 3.15. (Equality Elimination) can be proved in $\mathcal{H}$.

Proof. Recall the definition of equality $(\varphi_1 = \varphi_2) \equiv \lfloor\varphi_1 \leftrightarrow \varphi_2\rfloor$. Theorem 3.7 together with Proposition 3.5 give us a nice way to deal with equality premises. To prove $\vdash_{\mathcal{H}} (\varphi_1 = \varphi_2) \to (\psi[\varphi_1/x] \to \psi[\varphi_2/x])$, we apply Theorem 3.7 and prove $\{\varphi_1 \leftrightarrow \varphi_2\} \vdash_{\mathcal{H}} \psi[\varphi_1/x] \to \psi[\varphi_2/x]$, which is proved by Proposition 3.5. Note that the (formal) proof given in Proposition 3.5 does not use ($\exists$-Generalization) at all, so the conditions of Theorem 3.7 are satisfied. QED.

Lemma 3.16. (Functional Substitution) can be proved in $\mathcal{H}$.

Proof. Let z be a fresh variable that does not occur free in φ and φ', and is distinct from x. Notice the side condition that y does not occur free in φ'.

$$
\begin{array}{r|ll}
1 & \varphi' = z \leftrightarrow z = \varphi' & \text{definition} \\
2 & z = \varphi' \to (\varphi[z/x] \to \varphi[\varphi'/x]) & \text{Lemma 3.15} \\
3 & (\forall x \,.\, \varphi) \to \varphi[z/x] & \text{by axiom} \\
4 & \varphi' = z \to ((\forall x \,.\, \varphi) \to \varphi[z/x]) & \text{FOL reasoning} \\
5 & \varphi' = z \to (\varphi[z/x] \to \varphi[\varphi'/x]) & \text{FOL reasoning} \\
6 & \varphi' = z \to ((\forall x \,.\, \varphi) \to \varphi[\varphi'/x]) & \text{FOL reasoning} \\
7 & \forall z \,.\, (\varphi' = z \to ((\forall x \,.\, \varphi) \to \varphi[\varphi'/x])) & \text{by 6} \\
8 & (\exists z \,.\, \varphi' = z) \to ((\forall x \,.\, \varphi) \to \varphi[\varphi'/x]) & \text{FOL reasoning} \\
9 & (\forall x \,.\, \varphi) \land (\exists z \,.\, \varphi' = z) \to \varphi[\varphi'/x] & \text{FOL reasoning} \\
10 & (\forall x \,.\, \varphi) \land (\exists y \,.\, \varphi' = y) \to \varphi[\varphi'/x] & \text{FOL reasoning}
\end{array}
$$

QED.

Lemma 3.17. $\vdash_{\mathcal{H}} C_\sigma[\varphi_1 \land (x \in \varphi_2)] = C_\sigma[\varphi_1] \land (x \in \varphi_2)$.

Proof. We first prove $\vdash_{\mathcal{H}} C_\sigma[\varphi_1 \land (x \in \varphi_2)] \to C_\sigma[\varphi_1] \land (x \in \varphi_2)$. By FOL reasoning, it suffices to show both $\vdash_{\mathcal{H}} C_\sigma[\varphi_1 \land (x \in \varphi_2)] \to C_\sigma[\varphi_1]$ and $\vdash_{\mathcal{H}} C_\sigma[\varphi_1 \land (x \in \varphi_2)] \to (x \in \varphi_2)$. The first follows immediately by (Framing) and FOL reasoning. The second can be proved as:

$$
\begin{array}{r|l}
1 & \lceil x \rceil \\
2 & \lceil (x \land \neg\varphi_2) \lor (x \land \varphi_2) \rceil \\
3 & \lceil x \land \neg\varphi_2 \rceil \lor \lceil x \land \varphi_2 \rceil \\
4 & \neg\lceil x \land \neg\varphi_2 \rceil \to \lceil x \land \varphi_2 \rceil \\
5 & C_\sigma[\lceil x \land \varphi_2 \rceil] \to \neg\lceil x \land \neg\varphi_2 \rceil \\
6 & C_\sigma[\lceil x \land \varphi_2 \rceil] \to \lceil x \land \varphi_2 \rceil \\
7 & C_\sigma[\varphi_1 \land \lceil x \land \varphi_2 \rceil] \to C_\sigma[\lceil x \land \varphi_2 \rceil] \\
8 & C_\sigma[\varphi_1 \land \lceil x \land \varphi_2 \rceil] \to \lceil x \land \varphi_2 \rceil \\
9 & C_\sigma[\varphi_1 \land (x \in \varphi_2)] \to (x \in \varphi_2)
\end{array}
$$

QED.

Lemma 3.18. $\vdash_{\mathcal{H}} \exists y \,.\, ((x = y) \land \varphi) = \varphi[x/y]$ if x and y are distinct.

Proof. The proof is by structural induction on φ and Lemma 3.17. QED.

Lemma 3.19. $\vdash_{\mathcal{H}} \varphi = \exists y \,.\, (\lceil y \land \varphi \rceil \land y)$ if $y \notin \mathit{freeVar}(\varphi)$.

Proof. We first prove $\vdash_{\mathcal{H}} \exists y \,.\, (\lceil y \land \varphi \rceil \land y) \to \varphi$.

$$\begin{array}{r|ll}
1 & \neg(\lceil y \wedge \varphi \rceil \wedge (y \wedge \neg\varphi)) & (\textsc{Singleton Variable}) \\
2 & \lceil y \wedge \varphi \rceil \wedge y \to \varphi & \text{by 1, FOL reasoning} \\
3 & \forall y \,.\, (\lceil y \wedge \varphi \rceil \wedge y \to \varphi) & \text{by 2, axiom} \\
4 & \exists y \,.\, (\lceil y \wedge \varphi \rceil \wedge y) \to \varphi & \text{by 3, FOL reasoning}
\end{array}$$

We then prove $\vdash_{\mathcal{H}} \varphi \to \exists y \,.\, (\lceil y \wedge \varphi \rceil \wedge y)$. Let x be a fresh variable distinct from y.

$$\begin{array}{r|l}
1 & x \in \varphi \to x \in \varphi \\
2 & x \in \varphi \to \lceil x \wedge \varphi \rceil \\
3 & x \in \varphi \to \lceil x \wedge \lceil x \wedge \varphi \rceil \rceil \\
4 & x \in \varphi \to x \in \lceil x \wedge \varphi \rceil \\
5 & x \in \varphi \to \exists y \,.\, (x = y \wedge x \in \lceil y \wedge \varphi \rceil) \\
6 & x \in \varphi \to \exists y \,.\, (x \in y \wedge x \in \lceil y \wedge \varphi \rceil) \\
7 & x \in \varphi \to \exists y \,.\, (x \in (y \wedge \lceil y \wedge \varphi \rceil)) \\
8 & x \in \varphi \to x \in \exists y \,.\, (y \wedge \lceil y \wedge \varphi \rceil) \\
9 & x \in (\varphi \to \exists y \,.\, (y \wedge \lceil y \wedge \varphi \rceil)) \\
10 & \forall x \,.\, (x \in (\varphi \to \exists y \,.\, (y \wedge \lceil y \wedge \varphi \rceil))) \\
11 & \varphi \to \exists y \,.\, (y \wedge \lceil y \wedge \varphi \rceil)
\end{array}$$

QED.

Lemma 3.20. (Membership Symbol) is provable in $\mathcal{H}$.

Proof. We first prove $\vdash_{\mathcal{H}} x \in C_\sigma[\varphi] \to \exists y \,.\, (y \in \varphi \wedge x \in C_\sigma[y])$. Let $\Psi \equiv \exists y \,.\, (y \in \varphi \wedge x \in C_\sigma[y])$.

$$\begin{array}{r|l}
1 & \exists y \,.\, (y \in \varphi \wedge x \in C_\sigma[y]) \to \Psi \\
2 & \exists y \,.\, (\lceil y \wedge \varphi \rceil \wedge x \in C_\sigma[y]) \to \Psi \\
3 & \exists y \,.\, (\lceil x \wedge \lceil y \wedge \varphi \rceil \rceil \wedge x \in C_\sigma[y]) \to \Psi \\
4 & \exists y \,.\, (x \in \lceil y \wedge \varphi \rceil \wedge x \in C_\sigma[y]) \to \Psi \\
5 & \exists y \,.\, (x \in (\lceil y \wedge \varphi \rceil \wedge C_\sigma[y])) \to \Psi \\
6 & x \in \exists y \,.\, (\lceil y \wedge \varphi \rceil \wedge C_\sigma[y]) \to \Psi \\
7 & x \in \exists y \,.\, C_\sigma[\lceil y \wedge \varphi \rceil \wedge y] \to \Psi \\
8 & x \in C_\sigma[\exists y \,.\, \lceil y \wedge \varphi \rceil \wedge y] \to \Psi \\
9 & x \in C_\sigma[\varphi] \to \Psi
\end{array}$$

We then prove $\vdash_{\mathcal{H}} \exists y \,.\, (y \in \varphi \wedge x \in C[y]) \to x \in C[\varphi]$. In fact, we just need to apply the same derivation as above on $\vdash_{\mathcal{H}} \Psi \to \exists y \,.\, (y \in \varphi \wedge x \in C[y])$. QED.

So far, we have proved all the proof rules of $\mathcal{P}$ using $\mathcal{H}$ and the definedness axioms. Therefore, we have Lemma 3.21.

Lemma 3.21. Let Γ be a theory that contains the definedness symbols and axioms. For every pattern φ, $\Gamma \vdash_{\mathcal{P}} \varphi$ implies $\Gamma \vdash_{\mathcal{H}} \varphi$.

Therefore, $\mathcal{H}$ is complete for theories containing definedness.

Theorem 3.8. Let Γ be a theory that contains the definedness symbols and axioms. For every pattern φ, $\Gamma \vDash \varphi$ implies $\Gamma \vdash_{\mathcal{H}} \varphi$.

Proof. Use Lemma 3.21 and the completeness of $\mathcal{P}$ (Theorem 2.19). QED.

3.3 LOCAL COMPLETENESS

We will present and prove local completeness for $\mathcal{H}$. Local completeness states the equivalence between the local validity relation $\vDash^{loc}$ and the local provability relation $\vdash^{loc}_{\mathcal{H}}$. Both relations are stronger than their (global) counterparts $\vdash_{\mathcal{H}}$ and $\vDash$. The relation among these four relations has been shown in Figure 3.1.

Let us start by defining the two local relations.

Definition 3.3. Let Γ be a theory and φ be a pattern. The *local provability relation* $\Gamma \vdash^{loc}_{\mathcal{H}} \varphi$ holds iff there exists a finite subset $\Delta \subseteq \Gamma$ such that $\emptyset \vdash_{\mathcal{H}} \bigwedge \Delta \to \varphi$, where $\bigwedge \Delta$ is the conjunction of all patterns in Δ. We let $\bigwedge \emptyset$ be $\top$. The *local validity relation* $\Gamma \vDash^{loc} \varphi$ holds iff for any model M, any valuation ρ, and any element $a \in M$, $a \in |\psi|_{M,\rho}$ for all $\psi \in \Gamma$ implies $a \in |\varphi|_{M,\rho}$.

The local relations are stronger than their global counterparts. In addition, if $\Gamma = \emptyset$, the local relations are equivalent to their global counterparts.

Proposition 3.9. *For any Γ and φ, the following hold:*

1. $\Gamma \vdash^{loc}_{\mathcal{H}} \varphi$ *implies* $\Gamma \vdash_{\mathcal{H}} \varphi$*;*

2. $\Gamma \vDash^{loc} \varphi$ *implies* $\Gamma \vDash \varphi$*;*

3. $\emptyset \vdash^{loc}_{\mathcal{H}} \varphi$ *iff* $\emptyset \vdash_{\mathcal{H}} \varphi$*;*

4. $\emptyset \vDash^{loc} \varphi$ *iff* $\emptyset \vDash \varphi$.

Proof. (1). By definition, there exists a finite subset $\Delta \subseteq \Gamma$ such that $\emptyset \vdash_{\mathcal{H}} \bigwedge \Delta \to \varphi$. Note that $\Gamma \vdash_{\mathcal{H}} \bigwedge \Delta$, so by (Modus Ponens), $\Gamma \vdash_{\mathcal{H}} \varphi$.

(2). Let M be a model such that $M \vDash \Gamma$. Let ρ be any valuation and $a \in M$ be any element. Since $M \vDash \Gamma$, we have $M \vDash \psi$ for all $\psi \in \Gamma$. Therefore, $|\psi|_{M,\rho} = M$ for all $\psi \in \Gamma$, and thus $a \in |\psi|_{M,\rho}$ for all $\psi \in \Gamma$. By the definition of $\Gamma \vDash^{loc} \varphi$, $a \in |\varphi|_{M,\rho}$. Because a is arbitrarily chosen, $|\varphi|_{M,\rho} = M$. Because ρ is also arbitrarily chosen, $M \vDash \varphi$.

(3) and (4). By definition. QED.

We point out that the other directions of (1) and (2) in Proposition 3.9 do not hold in general. Consider $\Gamma = \{\neg x\}$. We will show that $\Gamma \nvdash^{loc}_{\mathcal{H}} \bot$ but $\Gamma \vdash_{\mathcal{H}} \bot$. To prove $\Gamma \nvdash^{loc}_{\mathcal{H}} \bot$, assume the opposite, that is, $\emptyset \vdash_{\mathcal{H}} \neg x \to \bot$. By the soundness of $\mathcal{H}$ (Theorem 3.1), $\emptyset \vDash \neg x \to \bot$. To show the contradiction, let us construct a model M whose carrier set is $\{0, 1\}$. Let ρ be a valuation such that $\rho(x) = 0$. Then we have $|\neg x \to \bot|_{M,\rho} = \{0\}$, which is not $\{0, 1\}$. This contradiction shows that $\Gamma \nvdash^{loc}_{\mathcal{H}} \bot$. On the other hand, we can prove $\Gamma \vdash_{\mathcal{H}} \bot$ as follows. Firstly, we have $\Gamma \vdash_{\mathcal{H}} \neg x$, which is equivalent to $\Gamma \vdash_{\mathcal{H}} x \to \bot$. By ($\exists$-Generalization), we have $\Gamma \vdash_{\mathcal{H}} (\exists x \,.\, x) \to \bot$. By (Existence) and (Modus Ponens), we have $\Gamma \vdash_{\mathcal{H}} \bot$. Therefore, the other directions of (1) and (2) in Proposition 3.9 do not hold in general.

Now, to establish Figure 3.1, we only need to prove local completeness, stated in Theorem 3.14. The proof presented here is inspired by the completeness proofs for polyadic modal logic [34] and hybrid logic [37], with novel techniques to handle sorts, many-sorted symbols, and quantifiers.

We start by defining consistent sets.

Definition 3.4. A theory Γ is *consistent*, if $\Gamma \nvdash^{loc}_{\mathcal{H}} \bot$. In addition, Γ is a *maximal consistent set* (MCS) if for every $\Gamma' \supsetneq \Gamma$, Γ' is inconsistent.

Intuitively, a consistent set gives a consistent "view" of elements in the underlying carrier set. Recall that a pattern is matched by certain elements. If Γ is consistent, all patterns in Γ can be matched by at least one common element. In other words, the infinite conjunction "pattern" $\bigwedge \Gamma$ is not $\bot$. The larger Γ is, the smaller $\bigwedge \Gamma$ becomes. An MCS is thus a maximal Γ, without making $\bigwedge \Gamma$ to be $\bot$. Note that the smallest patterns except $\bot$ are singleton patterns, which are matched by one element. Therefore, we can think of MCSs as representations of individual elements. This useful intuition motivates the definition of *canonical models* whose elements are MCSs (Definition 3.6) as well as the Truth Lemma (Lemma 3.26), which states that "Matching = Membership in MCSs". Truth Lemma is the key result that connects proofs and semantics.

We prove some properties about MCSs.

Proposition 3.10. *Let Γ be an MCS. The following propositions hold:*

1. *$\varphi \in \Gamma$ iff $\Gamma \vdash^{loc}_{\mathcal{H}} \varphi$; In particular, if $\vdash_{\mathcal{H}} \varphi$ then $\varphi \in \Gamma$;*

2. *$\neg\varphi \in \Gamma$ iff $\varphi \notin \Gamma$;*
3. *$\varphi_1 \wedge \varphi_2 \in \Gamma$ iff $\varphi_1 \in \Gamma$ and $\varphi_2 \in \Gamma$; In general, for any finite pattern set Δ, $\bigwedge \Delta \in \Gamma$ iff $\Delta \subseteq \Gamma$;*
4. *$\varphi_1 \vee \varphi_2 \in \Gamma$ iff $\varphi_1 \in \Gamma$ or $\varphi_2 \in \Gamma$; In general, for any finite pattern set Δ, $\bigvee \Delta \in \Gamma$ iff $\Delta \cap \Gamma \neq \emptyset$; As a convention, when $\Delta = \emptyset$, $\bigvee \Delta$ is $\bot$;*
5. *$\varphi_1, \varphi_1 \to \varphi_2 \in \Gamma$ implies $\varphi_2 \in \Gamma$; In particular, if $\vdash_{\mathcal{H}} \varphi_1 \to \varphi_2$, then $\varphi_1 \in \Gamma$ implies $\varphi_2 \in \Gamma$.*

Proof. By propositional reasoning. QED.

Definition 3.5. For an MCS Γ, we say that Γ is a *witnessed MCS*, if for every $\exists x \,.\, \varphi \in \Gamma$, there exists y such that $(\exists x \,.\, \varphi) \to \varphi[y/x] \in \Gamma$.

In the following, we show any consistent set Γ can be extended to a witnessed MCS Γ^+. The extension, however, requires an extension of the set of variables. To see why such an extension is needed, consider the following example. Let $\Sigma = (S, V, \Sigma)$ be a signature and $\Gamma = \{\neg x \mid x \in V\}$ be a pattern set containing all variable negations. We leave it for the readers to show that Γ is consistent. Here, we claim the consistent set Γ cannot be extended to a witnessed MCS Γ^+ in the signature Σ. The proof is by contradiction. Assume Γ^+ exists. By Proposition 3.10 and (Existence), $\exists x \,.\, x \in \Gamma^+$. Because Γ^+ is a witnessed MCS, there is a variable y such that $(\exists x \,.\, x) \to y \in \Gamma^+$, and by Proposition 3.10, $y \in \Gamma^+$. On the other hand, $\neg y \in \Gamma \subseteq \Gamma^+$. This contradicts the consistency of Γ^+.

Lemma 3.22. Let $\Sigma = (S, V, \Sigma)$ be a signature and Γ be a consistent set. Extend the variable set V to V^+ with countably infinitely many new variables, and denoted the extended signature as $\Sigma^+ = (V^+, S, \Sigma)$. There exists a pattern set Γ^+ in the extended signature Σ^+ such that $\Gamma \subseteq \Gamma^+$ and Γ^+ is a witnessed MCS.

Proof. We use MLPattern and MLPattern$^+$ denote the set of all patterns in the original and extended signatures, respectively. Enumerate all patterns $\varphi_1, \varphi_2, \dots \in$ MLPattern$^+$ and all variables $\mathbb{x}_1, \mathbb{x}_2, \dots$ in $V^+ \setminus V$. We will construct a non-decreasing sequence of pattern sets $\Gamma_0 \subseteq \Gamma_1 \subseteq \Gamma_2 \dots \subseteq$ MLPattern^+_s, with $\Gamma_0 = \Gamma$. Notice that Γ_0 contains variables only in V. Eventually, we will let $\Gamma^+ = \bigcup_{i \geq 0} \Gamma_i$ to be the witnessed MCS.

For every $n \geq 1$, we define Γ_n as follows. If $\Gamma_{n-1} \cup \{\varphi_n\}$ is inconsistent, then let $\Gamma_n = \Gamma_{n-1}$. Otherwise,

if φ_n is not of the form $\exists x \,.\, \psi$:

$$\begin{aligned}
&\qquad \Gamma_n = \Gamma_{n-1} \cup \{\varphi_n\} \\
&\text{if } \varphi_n \equiv \exists x \,.\, \psi \text{ and } \mathbb{x}_i \text{ is the first variable in } V^+ \setminus V \\
&\text{that does not occur free in } \Gamma_{n-1} \text{ and } \psi\text{:} \\
&\qquad \Gamma_n = \Gamma_{n-1} \cup \{\varphi_n\} \cup \{\psi[\mathbb{x}_i/x]\}
\end{aligned}$$

Notice that in the second case, we can always pick a variable $\mathbb{x}_i$ that satisfies the conditions because by construction, $\Gamma_{n-1} \cup \{\varphi_n\}$ uses at most finitely many variables in $V^+ \setminus V$.

We show that Γ_n is consistent for every $n \geq 0$ by induction. The base case is to show Γ_0 is consistent in the extended signature. Assume it is not. Then there exists a finite subset $\Delta_0 \subseteq_{\text{fin}} \Gamma_0$ such that $\vdash_{\mathcal{H}} \bigwedge \Delta_0 \to \bot$. The proof of $\bigwedge \Delta_0 \to \bot$ is a finite sequence of patterns in MLPATTERN$^+$. We can replace every occurrence of the variable $\mathbb{y} \in V^+ \setminus V$ ($\mathbb{y}$ can have any sort) with a variable $y \in V$ that has the same sort as $\mathbb{y}$ and does not occur (no matter bound or free) in the proof. By induction on the length of the proof, the resulting sequence is also a proof of $\bigwedge \Delta_0 \to \bot$, and it consists of only patterns in PATTERN. This contradicts the consistency of Γ_0 as a subset of PATTERN, and this contradiction finishes our proof of the base case.

Now assume Γ_{n-1} is consistent for $n \geq 1$. We will show Γ_n is also consistent. If $\Gamma_{n-1} \cup \{\varphi_n\}$ is inconsistent or φ_n does not have the form $\exists x \,.\, \psi$, Γ_n is consistent by construction. Assume $\Gamma_{n-1} \cup \{\varphi_n\}$ is consistent, $\varphi_n \equiv \exists x \,.\, \psi$, but $\Gamma_n = \Gamma_{n-1} \cup \{\varphi_n\} \cup \{\psi[\mathbb{x}_i/x]\}$ is not consistent. Then there exists a finite subset $\Delta \subseteq_{\text{fin}} \Gamma_{n-1} \cup \{\varphi_n\}$ such that $\vdash_{\mathcal{H}} \bigwedge \Delta \to \neg\psi[\mathbb{x}_i/x]$. By (UNIVERSAL GENERALIZATION), $\vdash_{\mathcal{H}} \forall \mathbb{x}_i \,.\, (\bigwedge \Delta \to \neg\psi[\mathbb{x}_i/x])$. Notice that $\mathbb{x}_i \notin \mathit{freeVar}(\bigwedge \Delta)$ by construction, so by FOL reasoning $\vdash_{\mathcal{H}} \bigwedge \Delta \to \neg\exists \mathbb{x}_i \,.\, (\psi[\mathbb{x}_i/x])$. Since $\mathbb{x}_i \notin \mathit{freeVar}(\psi)$, by α-renaming, $\exists \mathbb{x}_i \,.\, (\psi[\mathbb{x}_i/x]) \equiv \exists x \,.\, \psi \equiv \varphi_n$, and thus $\vdash_{\mathcal{H}} \bigwedge \Delta \to \neg\varphi_n$. This contradicts the assumption that $\Gamma_{n-1} \cup \{\varphi_n\}$ is consistent.

Since Γ_n is consistent for any $n \geq 0$, $\Gamma^+ = \bigcup_n \Gamma_n$ is also consistent. This is because the derivation that shows inconsistency would use only finitely many patterns in Γ^+. In addition, we know Γ^+ is maximal and witnessed by construction. QED.

We will prove that for every witnessed MCS $\Gamma = \{\Gamma_s\}_{s \in S}$, there exists a model M and a valuation ρ such that for every $\varphi \in \Gamma_s$, $|\varphi|_{M,\rho} \neq \emptyset$. The next definition defines the canonical model which contains all witnessed MCSs as its elements. We will construct our intended model M as a submodel of the canonical model.

Definition 3.6. Given a signature $\Sigma = (S, \Sigma)$. The canonical model $W = (\{W_s\}_{s \in S}, \_W)$ consists of

- a carrier set $W_s = \{\Gamma \mid \Gamma \text{ is a witnessed MCS of sort } s\}$ for every sort $s \in S$;

- an interpretation $\sigma_W \colon W_{s_1} \times \cdots \times W_{s_n} \to \mathcal{P}(W_s)$ for every symbol $\sigma \in \Sigma_{s_1 \dots s_n, s}$, defined as $\Gamma \in \sigma_W(\Gamma_1, \dots, \Gamma_n)$ if and only if for any $\varphi_i \in \Gamma_i$, $1 \le i \le n$, $\sigma(\varphi_1, \dots, \varphi_n) \in \Gamma$; In particular, the interpretation for a constant symbol $\sigma \in \Sigma_{\lambda,s}$ is $\sigma_W = \{\Gamma \in W_s \mid \sigma \in \Gamma\}$.

The carrier set W is not empty, thanks to Lemma 3.22.

The canonical model has a nontrivial property stated as the next lemma. The proof of the lemma is difficult, so we leave it to the end of the subsection.

Theorem 3.11. Let $\Sigma = (S, \Sigma)$ be a signature and Γ be a witnessed MCS of sort $s \in S$. Given a symbol $\sigma \in \Sigma_{s_1 \dots s_n, s}$ and patterns $\varphi_1, \dots, \varphi_n$ of appropriate sorts. If $\sigma(\varphi_1, \dots, \varphi_n) \in \Gamma$, then there exist n witnessed MCSs $\Gamma_1, \dots, \Gamma_n$ of appropriate sorts such that $\varphi_i \in \Gamma_i$ for every $1 \le i \le n$, and $\Gamma \in \sigma_W(\Gamma_1, \dots, \Gamma_n)$.

Definition 3.7. Let $\Sigma = (S, \Sigma)$ be a signature and $W = (\{W_s\}_{s \in S}, \_W)$ be the canonical model. Given a witnessed MCS $\Gamma = \{\Gamma_s\}_{s \in S}$. Define $Y = \{Y_s\}_{s \in S}$ be the smallest sets such that $Y_s \subseteq W_s$ for every sort s, and the following inductive properties are satisfied:

- $\Gamma_s \in Y_s$ for every sort s;
- If $\Delta \in Y_s$ and there exist a symbol $\sigma \in \Sigma_{s_1 \dots s_n, s}$ and witnessed MCSs $\Delta_1, \dots, \Delta_n$ of appropriate sorts such that $\Delta \in \sigma_W(\Delta_1, \dots, \Delta_n)$, then $\Delta_1 \in Y_{s_1}, \dots, \Delta_n \in Y_{s_n}$.

Let $Y = (Y, \_Y)$ be the model generated from Γ, where

$$\sigma_Y(\Delta_1, \dots, \Delta_n) = Y_s \cap \sigma_W(\Delta_1, \dots, \Delta_n) \quad \text{for every } \sigma \in \Sigma_{s_1 \dots s_n, s} \text{ and } \Delta_1 \in Y_{s_1}, \dots, \Delta_n \in Y_{s_n}.$$

We give some intuition about the generated model $Y = (Y, \_Y)$. The interpretation σ_Y is just the restriction of the interpretation σ_M on Y. The carrier set Y is defined inductively. Firstly, Y contains Γ. Given a set $\Delta \in Y$. If sets $\Delta_1, \dots, \Delta_n$ are “generated” from Δ by a symbol σ, meaning that $\Delta \in \sigma_W(\Delta_1, , \dots, \Delta_n)$, then they are also in Y. Of course, a set Δ is in Y maybe because it is generated from a set Δ' by a symbol σ', while Δ' is generated from a set Δ'' by a symbol σ'', and so on. This generating path keeps going and eventually ends at Γ in finite number of steps. By definition, every member of Y has at least one such generating path, which we formally define as follows.

Definition 3.8. Let $\Gamma = \{\Gamma_s\}_{s \in S}$ be a witnessed MCS and Y be the model generated from Γ. A *generating path* π is either the empty path ϵ, or a sequence of pairs $\langle (\sigma_1, p_1), \dots, (\sigma_k, p_k) \rangle$ where $\sigma_1, \dots, \sigma_k$ are symbols (not necessarily distinct) and $p_1, \dots, p_k$ are natural numbers representing positions. The *generating path relation*, denoted as GP, is a binary relation between witnessed MCSs in Y and generating paths, defined as the smallest relation that satisfies the following conditions:

- $GP(\Gamma_s, \epsilon)$ holds for every sort s;
- If $GP(\Delta, \pi)$ holds for a set $\Delta \in Y_s$ and a generating path π, and there exist a symbol $\sigma \in \Sigma_{s_1 \dots s_n, s}$ and witnessed MCSs $\Delta_1, \dots, \Delta_n$ such that $\Delta \in \sigma_W(\Delta_1, \dots, \Delta_n)$, then $GP(\Delta_i, \langle \pi, (\sigma, i) \rangle)$ holds for every $1 \leq i \leq n$.

We say that Δ has a generating path π in the generated model if $GP(\Delta, \pi)$ holds. It is easy to see that every witnessed MCS in Y has at least one generating path, and if a witnessed MCS of sort s has the empty path ϵ as its generating path, it must be Γ_s itself.

Definition 3.9. Given a generating path π. Define the application context C_π inductively as follows. If $\pi = \epsilon$, then C_π is the identity context $\square$. If $\pi = \langle \pi_0, (\sigma, i) \rangle$ where $\sigma \in \Sigma_{s_1 \dots s_n, s}$ and $1 \leq i \leq n$, then $C_\pi = C_{\pi_0}[\sigma(\top_{s_1}, \dots, \top_{s_{i-1}}, \square, \top_{s_{i+1}}, \dots, \top_{s_n})]$.

A good intuition about Definition 3.9 is given as the next lemma.

Lemma 3.23. Let Γ be a witnessed MCS and Y be the model generated from Γ. Let $\Delta \in Y$. If Δ has a generating path π, then $C_\pi[\varphi] \in \Gamma$ for any pattern $\varphi \in \Delta$.

Proof. The proof is by induction on the length of the generating path π. If π is the empty path ϵ, then Δ must be Γ and C_π is the identity context, and $C_\pi[\varphi] = \varphi \in \Gamma$ for any $\varphi \in \Delta$. Now assume Δ has a generating path $\pi = \langle \pi_0, (\sigma, i) \rangle$ with $\sigma \in \Sigma_{s_1 \dots s_n, s}$. By Definition 3.8, there exist witnessed MCSs $\Delta_{s_1}, \dots, \Delta_{s_n}, \Delta_s \in Y$ and $1 \leq i \leq n$ such that $\Delta = \Delta_{s_i}$, $\Delta_s \in \sigma_W(\Delta_{s_1}, \dots, \Delta_{s_n})$, and Δ_s has π_0 as its generating path. For every $\varphi \in \Delta = \Delta_i$, since $\top_{s_j} \in \Delta_{s_j}$ for any $j \neq i$, by Definition 3.6, $\sigma(\top_{s_1}, \dots, \top_{s_{i-1}}, \varphi, \top_{s_{i+1}}, \dots, \top_{s_n}) \in \Delta_s$. By induction hypothesis, $C_{\pi_0}[\sigma(\top_{s_1}, \dots, \top_{s_{i-1}}, \varphi, \top_{s_{i+1}}, \dots, \top_{s_n})] \in \Gamma$, while the latter is exactly $C_\pi[\varphi]$. QED.

Lemma 3.24. Let Γ be a witnessed MCS and Y be the model generated from Γ. For every $\Gamma_1, \Gamma_2 \in Y$ of the same sort and every variable x, if $x \in \Gamma_1 \cap \Gamma_2$ then $\Gamma_1 = \Gamma_2$.

Proof. Let π_i be a generating path of Γ_i for $i = 1, 2$. Assume $\Gamma_1 \neq \Gamma_2$. Then there exists a pattern φ such that $\varphi \in \Gamma_1$ and $\neg\varphi \in \Gamma_2$. Because $x \in \Gamma_1 \cap \Gamma_2$, we know $x \wedge \varphi \in \Gamma_1$ and $x \wedge \neg\varphi \in \Gamma_2$. By Lemma 3.23, $C_{\pi_1}[x \wedge \varphi], C_{\pi_2}[x \wedge \neg\varphi] \in \Gamma$, and thus $C_{\pi_1}[x \wedge \varphi] \wedge C_{\pi_2}[x \wedge \neg\varphi] \in \Gamma$. On the other hand, $\neg(C_{\pi_1}[x \wedge \varphi] \wedge C_{\pi_2}[x \wedge \neg\varphi])$ is an instance of (Singleton Variable) and thus it is included in Γ. This contradicts the consistency of Γ. QED.

We will establish an important result about generated models in Lemma 3.26 (the Truth Lemma), which links the semantics and syntax and is essential to the completeness result. Roughly speaking, the lemma says that for any generated model Y and any witnessed MCS

$\Delta \in Y$, a pattern φ is in Δ if and only if the interpretation of φ in Y contains Δ. To prove the lemma, it is important to show that every variable is interpreted to a singleton. Lemma 3.24 ensures that every variable belongs to at most one witnessed MCS. To make sure it is interpreted to exactly one MCS, we complete our model by adding a dummy element $\star$ to the carrier set, and interpreting all variables which are interpreted to none of the MCSs to the dummy element. This motivates the next definition.

Definition 3.10. Let $\Gamma = \{\Gamma_s\}_{s \in S}$ be a witnessed MCS and Y be the Γ-generated model. Γ-*completed model*, denoted as $M = (\{M_s\}_{s \in S}, \_M)$, is inductively defined as follows for all sorts $s \in S$:

- $M_s = Y_s$, if every $x : s$ belongs at least one MCS in Y_s;
- $M_s = Y_s \cup \{\star_s\}$, otherwise.

We assume $\star_s$ is an entity that is different from any MCSs, and $\star_{s_1} \neq \star_{s_2}$ if $s_1 \neq s_2$. For every $\sigma \in \Sigma_{s_1 \ldots s_n, s}$, define its interpretation

$$\sigma_M(\Delta_1, \ldots, \Delta_n) = \begin{cases} \emptyset & \text{if some } \Delta_i = \star_{s_i} \\ \sigma_Y(\Delta_1, \ldots, \Delta_n) \cup \{\star_s\} & \text{if all } \Delta_j \neq \star_{s_j} \text{ and some } \Delta_i = \Gamma_{s_i} \\ \sigma_{\mathcal{Y}_{\Gamma_0}}(\Delta_1, \ldots, \Delta_n) & \text{otherwise} \end{cases}$$

The completed valuation $\rho \colon V \to M$ is defined as

$$\rho(x : s) = \begin{cases} \Delta & \text{if } x : s \in \Delta \\ \star_s & \text{otherwise} \end{cases}$$

The valuation ρ is a well-defined function, because by Lemma 3.24, if there are two witnessed MCSs Δ_1 and Δ_2 such that $x \in \Delta_1$ and $x \in \Delta_2$, then $\Delta_1 = \Delta_2$.

Now we come back to prove Lemma 3.11. We need the following technical lemma.

Lemma 3.25. Let $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ be a symbol, $\Phi_1, \ldots, \Phi_n, \phi$ be patterns of appropriate sorts, and $y_1, \ldots, y_n, x$ be variables of appropriate sorts such that $y_1, \ldots, y_n$ are distinct, and

$$y_1, \ldots, y_n \notin \mathit{freeVar}(\phi) \cup \bigcup_{1 \le i \le n} \mathit{freeVar}(\Phi_i)$$

Then we have

$$\vdash \sigma(\Phi_1, \ldots, \Phi_n) \to \exists y_1 \ldots \exists y_n \,.\, \sigma(\Phi_1 \wedge (\exists x \,.\, \phi \to \phi[y_1/x]), \ldots, \Phi_n \wedge (\exists x \,.\, \phi \to \phi[y_n/x]))$$

Proof. Notice that for every $1 \le i \le n$,

$$\vdash_{\mathcal{H}} \exists x \,.\, \phi \to \exists y_i.(\phi[y_i/x]).$$

By easy matching logic reasoning,

$$\vdash \sigma(\Phi_1, \ldots, \Phi_n) \to \sigma(\Phi_1 \wedge (\exists x \,.\, \phi \to \exists y_1.(\phi[y_1/x])), \ldots, \Phi_n \wedge (\exists x \,.\, \phi \to \exists y_n.(\phi[y_n/x])))$$

Then use Proposition 3.4 to move the quantifiers $\exists y_1, \ldots, \exists y_n$ to the top. QED.

Now we are ready to prove Lemma 3.11.

Proof of Lemma 3.11. Recall that $\Gamma \in \sigma_W(\Gamma_1, \ldots, \Gamma_n)$ means for every $\phi_i \in \Gamma_i$, $1 \le i \le n$, $\sigma(\phi_1, \ldots, \phi_n) \in \Gamma$. The main technique that we will be using here is similar to Lemma 3.22. We start with the singleton sets $\{\varphi_i\}$ for every $1 \le i \le n$ and extend them to witnessed MCSs Γ_i, while this time we also need to make sure the results $\Gamma_1, \ldots, \Gamma_n$ satisfy the desired property $\Gamma \in \sigma_W(\Gamma_1, \ldots, \Gamma_n)$. Another difference compared to Lemma 3.22 is that this time we do not extend our set of variables, because our starting point, $\{\varphi_i\}$, contains just one pattern and uses only finitely many variables. Readers will see how these conditions play a role in the upcoming proof.

Enumerate all patterns of sorts $s_1, \ldots, s_n$ as follows $\psi_0, \psi_1, \psi_2, \cdots \in \bigcup_{1 \le i \le n} \textsc{Pattern}_{s_i}$. Notice that $s_1, \ldots, s_n$ do not need to be all distinct. To ease our notation, we define a "choice" operator, denoted as $[\varphi_s]_{s'}$, as follows

$$[\varphi_s]_{s'} = \begin{cases} \varphi_s & \text{if } s = s' \\ \text{nothing} & \text{otherwise} \end{cases}$$

For example, $\varphi_s \wedge [\psi]_s$ means $\varphi_s \wedge \psi$ if ψ also has sort s. Otherwise, it means φ_s. The choice operator propagates with all logic connectives in the natural way. For example, $[\neg\psi]_s = \neg[\psi]_s$.

In the following, we will define a non-decreasing sequence of pattern sets $\Gamma_i^{(0)} \subseteq \Gamma_i^{(1)} \subseteq \Gamma_i^{(2)} \subseteq \cdots \subseteq \textsc{Pattern}_{s_i}$ for each $1 \le i \le n$, such that the following conditions are true for all $1 \le i \le n$ and $k \ge 0$:

1. If ψ_k has sort s_i, then either ψ_k or $\neg\psi_k$ belongs to $\Gamma_i^{(k+1)}$.

2. If ψ_k has the form $\exists x \,.\, \phi_k$ and it belongs to $\Gamma_i^{(k+1)}$, then there exists a variable z such that $(\exists x \,.\, \phi_k) \to \phi_k[z/x]$ also belongs to $\Gamma_i^{(k+1)}$.

3. $\Gamma_i^{(k)}$ is finite.

4. Let $\pi_i^{(k)} = \bigwedge \Gamma_i^{(k)}$ for every $1 \leq i \leq n$. Then $\sigma(\pi_1^{(k)}, \dots, \pi_n^{(k)}) \in \Gamma$.

5. $\Gamma_i^{(k)}$ is consistent.

Among the above five conditions, condition (2)–(5) are like "safety" properties while condition (1) is like a "liveness" properties. We will eventually let $\Gamma_i = \bigcup_{k \geq 0} \Gamma_i^{(k)}$ and prove that Γ_i has the desired property. Before we present the actual construction, we give some hints on how to prove these conditions hold. Conditions (1)–(3) will be satisfied directly by construction, although we will put a notable effort in satisfying condition (2). Condition (4) will be proved hold by induction on k. Condition (5) is in fact a consequence of condition (4) as shown below. Assume condition (4) holds but condition (5) fails. This means that $\Gamma_i^{(k)}$ is not consistent for some $1 \leq i \leq n$, so $\vdash_{\mathcal{H}} \pi_i^{(k)} \to \bot$. By (Framing)

$$\vdash_{\mathcal{H}} \sigma(\pi_1^{(k)}, \dots, \pi_i^{(k)}, \dots, \pi_n^{(k)}) \to \sigma(\pi_1^{(k)}, \dots, \bot, \dots, \pi_n^{(k)})$$

Then by Proposition 3.4 and FOL reasoning,

$$\vdash_{\mathcal{H}} \sigma(\pi_1^{(k)}, \dots, \pi_i^{(k)}, \dots, \pi_n^{(k)}) \to \bot$$

Since $\sigma(\pi_1^{(k)}, \dots, \pi_i^{(k)}, \dots, \pi_n^{(k)}) \in \Gamma$ by condition (4), we know $\bot \in \Gamma$ by Proposition 3.10. And this contradicts the fact that Γ is consistent.

Now we are ready to construct the sequence $\Gamma_i^{(0)} \subseteq \Gamma_i^{(1)} \subseteq \Gamma_i^{(2)} \subseteq \dots$ for $1 \leq i \leq n$. Let $\Gamma_i^{(0)} = \{\varphi_i\}$ for $1 \leq i \leq n$. Obviously, $\Gamma_i^{(0)}$ satisfies conditions (3) and (4). Condition (5) follows as a consequence of condition (4). Conditions (1) and (2) are not applicable.

Suppose we have already constructed sets $\Gamma_i^{(k)}$ for every $1 \leq i \leq n$ and $k \geq 0$, which satisfy the conditions (1)–(5). We show how to construct $\Gamma_i^{(k+1)}$. In order to satisfy condition (1), we should add either ψ_k or $\neg\psi_k$ to $\Gamma_i^{(k)}$, if $\Gamma_i^{(k)}$ has the same sort as ψ_k. Otherwise, we simply let $\Gamma_i^{(k+1)}$ be the same as $\Gamma_i^{(k)}$. The question here is: if $\Gamma_i^{(k)}$ has the same sort as ψ_k, which pattern should we add to $\Gamma_i^{(k)}$, ψ_k or $\neg\psi_k$? Obviously, condition (3) will still hold no matter which one we choose to add, so we just need to make sure that we do not break conditions (2) and (4).

Let us start by satisfying condition (4). Consider pattern $\sigma(\pi_1^{(k)}, \dots, \pi_n^{(k)})$, which, by condition (4), is in Γ. This tells us that the pattern

$$\sigma(\pi_1^{(k)} \wedge [\psi_k \vee \neg\psi_k]_{s_1}, \dots, \pi_n^{(k)} \wedge [\psi_k \vee \neg\psi_k]_{s_n})$$

is also in Γ. Recall that $[\_]_s$ is the choice operator, so if ψ_k has sort s_i, then $\pi_i^{(k)} \wedge [\psi_k \vee \neg\psi_k]_{s_i}$ is $\pi_i^{(k)} \wedge (\psi_k \vee \neg\psi_k)$. Otherwise, it is $\pi_i^{(k)}$. Use Proposition 3.4 and FOL reasoning, and notice that the choice operator propagates with the disjunction $\vee$ and the negation $\neg$, we get

$$\sigma((\pi_1^{(k)} \wedge [\psi_k]_{s_1}) \vee (\pi_1^{(k)} \wedge \neg[\psi_k]_{s_1}), \ldots, (\pi_n^{(k)} \wedge [\psi_k]_{s_n}) \vee (\pi_n^{(k)} \wedge \neg[\psi_k]_{s_n})) \in \Gamma$$

Then we use Proposition 3.4 again and move all the disjunctions to the top, and we end up with a disjunction of 2^n patterns:

$$\bigvee \sigma(\pi_1^{(k)} \wedge [\neg]_1^{(k)}[\psi_k]_{s_1}, \ldots, \pi_n^{(k)} \wedge [\neg]_n^{(k)}[\psi_k]_{s_n}) \in \Gamma$$

where $[\neg]$ means either nothing or $\neg$. Notice that some $[\psi_k]_{s_i}$'s might be nothing, so some of these 2^n patterns may be the same.

Notice that Γ is an MCS. By proposition 3.10, among these 2^n patterns there must exists one pattern that is in Γ. We denote *that* pattern as

$$\sigma(\pi_1^{(k)} \wedge [\neg]_1^{(k)}[\psi_k]_{s_1}, \ldots, \pi_n^{(k)} \wedge [\neg]_n^{(k)}[\psi_k]_{s_n})$$

For any $1 \leq i \leq n$, if $[\neg]_i^{(k)}[\psi_k]_{s_i}$ does not have the form $\exists x \,.\, \phi$, we simply define $\Gamma_i^{(k+1)} = \Gamma_i^{(k)} \cup \{[\neg]_i^{(k)}[\psi_k]_{s_i}\}$. If $[\neg]_i^{(k)}[\psi_k]_{s_i}$ does have the form $\exists x \,.\, \phi$, we need special effort to satisfy condition (2). Without loss of generality and to ease our notation, let us assume that *for every* $1 \leq i \leq n$, the pattern $[\neg]_i^{(k)}[\psi_k]_{s_i}$ has the same form $\exists x \,.\, \phi$. We are going to find for each index i a variable z_i such that

$$\sigma(\pi_1^{(k)} \wedge \exists x \,.\, \phi \wedge (\exists x \,.\, \phi \to \phi[z_1/x]), \ldots, \pi_n^{(k)} \wedge \exists x \,.\, \phi \wedge (\exists x \,.\, \phi \to \phi[z_n/x])) \in \Gamma$$

This will allow us to define $\Gamma_i^{(k+1)} = \Gamma_i^{(k)} \cup \{\exists x \,.\, \phi\} \cup \{\exists x \,.\, \phi \to \phi[z_i/x]\}$ which satisfies conditions (2) and (4).

We find these variables z_i's by Lemma 3.25 and the fact that Γ is a witnessed set. Let $\Phi_i \equiv \pi_i^{(k)} \wedge \exists x \,.\, \phi$ for $1 \leq i \leq n$. By construction, $\sigma(\Phi_1, \ldots, \Phi_n) \in \Gamma$. Hence, by Lemma 3.25 and Proposition 3.10, for any distinct variables $y_1, \ldots, y_n \notin \mathit{freeVar}(\phi) \cup \bigcup_{1 \leq i \leq n} \mathit{freeVar}(\Phi_i)$,

$$\exists y_1 \ldots \exists y_n \,.\, \sigma(\Phi_1 \wedge (\exists x \,.\, \phi \to \phi[y_1/x]), \ldots, \Phi_n \wedge (\exists n \,.\, \phi \to \phi[y_n/x])) \in \Gamma$$

The set Γ is a witnessed set, so there exist variables $z_1, \ldots, z_n$ such that

$$\sigma(\Phi_1 \wedge (\exists x \,.\, \phi \to \phi[z_1/x]), \ldots, \Phi_n \wedge (\exists x \,.\, \phi \to \phi[z_n/x])) \in \Gamma$$

This justifies our construction $\Gamma_i^{(k+1)} = \Gamma_i^{(k)} \cup \{\exists x \,.\, \phi\} \cup \{\exists x \,.\, \phi \to \phi[z_i/x]\}$.

So far we have proved our construction of the sequences $\Gamma_i^{(0)} \subseteq \Gamma_i^{(1)} \subseteq \Gamma_i^{(2)} \subseteq \ldots$ for

$1 \leq i \leq n$ satisfy the conditions (1)–(5). Let $\Gamma_i = \bigcup_{k\geq 0} \Gamma_i^{(k)}$ for $1 \leq i \leq n$. By construction, Γ_i is a witnessed MCS. It remains to prove that $\Gamma \in \sigma_W(\Gamma_1, \ldots, \Gamma_n)$. To prove it, assume $\phi_i \in \Gamma_i$ for $1 \leq i \leq n$. By construction, there exists $K > 0$ such that $\phi_i \in \Gamma_i^{(K)}$ for all $1 \leq i \leq n$. Therefore, $\vdash_{\mathcal{H}} \pi_i^{(K)} \to \phi_i$. By condition (4), $\sigma(\pi_1^{(K)}, \ldots, \pi_n^{(K)}) \in \Gamma$, and thus by (Framing) and Proposition 3.10, $\sigma(\phi_1, \ldots, \phi_n) \in \Gamma$. QED.

Lemma 3.26 (Truth Lemma)**.** Let Γ be a witnessed MCS, M be its completed model, and ρ be the completed valuation. For any witnessed MCS $\Delta \in M$ and any pattern φ such that Δ and φ have the same sort,

$$\varphi \in \Delta \quad \text{if and only if} \quad \Delta \in |\varphi|_{M,\rho}$$

Proof. The proof is by induction on the structure of φ. If φ is a variable the conclusion follows by Definition 3.6. If φ has the form $\psi_1 \wedge \psi_2$ or $\neg\psi_1$, the conclusion follows from Proposition 3.10. If φ has the form $\sigma(\varphi_1, \ldots, \varphi_n)$, the conclusion from left to right is given by Lemma 3.11. The conclusion from right to left follows from Definition 3.6.

Now assume φ has the form $\exists x \,.\, \psi$. If $\exists x \,.\, \psi \in \Delta$, since Δ is a witnessed set, there is a variable y such that $\exists x \,.\, \psi \to \psi[y/x] \in \Delta$, and thus $\psi[y/x] \in \Delta$. By induction hypothesis, $\Delta \in |\psi[y/x]|_{M,\rho}$, and thus $\Delta \in |\exists x \,.\, \psi|_{M,\rho}$.

Consider the other direction. Assume $\Delta \in |\exists x \,.\, \psi|_{M,\rho}$. By definition there exists a witnessed set $\Delta' \in M$ such that $\Delta \in |\psi|_{M,\rho[\Delta'/x]}$. By Definition 3.10, every element in M (no matter if it is an MCS or $\star$) has a variable that is assigned to it by the completed valuation ρ. Let us assume that variable y is assigned to Δ', i.e., $\rho(y) = \Delta'$. By Lemma 3.1, $\Delta \in |\psi|_{M,\rho'} = |\psi[y/x]|_{M,\rho}$. By induction hypothesis, $\psi[y/x] \in \Delta$. Finally notice that $\vdash_{\mathcal{H}} \psi[y/x] \to \exists y \,.\, \psi[y/x]$. By Proposition 3.10, $\exists y \,.\, \psi[y/x] \in \Delta$, i.e., $\exists x \,.\, \psi \in \Delta$. QED.

Theorem 3.12. For any consistent set Γ, there is a model M and a valuation ρ such that for all patterns $\varphi \in \Gamma$, $|\varphi|_{M,\rho} \neq \emptyset$.

Proof. Use Lemma 3.22 and extend Γ to a witnessed MCS Γ^+. Let M and ρ be the completed model and valuation generated by Γ^+ respectively. By Lemma 3.26, for all patterns $\varphi \in \Gamma \subseteq \Gamma^+$, we have $\Gamma^+ \in |\varphi|_{M,\rho}$, so $|\varphi|_{M,\rho} \neq \emptyset$. QED.

Now we are ready to prove local completeness of $\mathcal{H}$.

Theorem 3.13. For any Γ and φ, $\Gamma \vDash^{loc} \varphi$ implies $\Gamma \vdash_{\mathcal{H}}^{loc} \varphi$.

Proof. Assume the opposite that $\Gamma \nvdash_{\mathcal{H}}^{loc} \varphi$, which implies that $\Gamma \cup \{\neg\varphi\}$ is consistent. Extend it to a witnessed MCS Γ^+ and let M, ρ be the completed model and completed valuation generated by Γ^+. By Lemma 3.26, $\Gamma^+ \in |\psi|_{M,\rho}$ for all $\psi \in \Gamma$, and $\Gamma^+ \in |\neg\varphi|_{M,\rho}$, i.e., $\Gamma^+ \notin |\varphi|_{M,\rho}$. This contradicts with $\Gamma \vDash^{loc} \varphi$. QED.

Theorem 3.14. For any φ, $\emptyset \vDash \varphi$ implies $\emptyset \vdash_{\mathcal{H}} \varphi$.

Proof. By Proposition 3.9. QED.

In the literature, both Theorem 3.13 and Theorem 3.14 are called local completeness. To distinguish them, Theorem 3.13 is called strong local completeness while Theorem 3.14 is called weak local completeness.

Chapter 4: FROM MATCHING LOGIC TO MATCHING μ-LOGIC

Fixpoints are ubiquitous in computer science. They are given different names when appearing in different contexts. Inductive datatypes are an example of fixpoints. The datatype of cons-lists `list ::= Nil | Cons(element, list)` is the smallest set that is closed under `Nil` and `Cons`. Many temporal operators are fixpoints; $\Box\varphi$ ("always") can be defined as a greatest fixpoint based on $\circ\varphi$ ("next"); see Section 5.8. The reachability relation $\varphi_1 \Rightarrow \varphi_2$ in reachability logic (RL) (Section 2.14) is also a fixpoint; see Section 5.11. Furthermore, these fixpoints are studied and reasoned about using different methods. For inductive datatypes, we often use structural induction to prove recursive properties about them. For temporal operators, we can use the specialized proof rules of temporal logics to reason about them. For example, $\Box(\varphi \to \circ\varphi) \to (\varphi \to \Box\varphi)$ is the (IND) proof rule of infinite-trace LTL (Figure 2.5) that captures the inductive nature of $\Box$. For RL, the (Circularity) proof rule in Figure 2.12 captures the co-inductive nature of reachability reasoning.

On the other hand, the Knaster-Tarski fixpoint theorem (Theorem 2.1) governs everything we need to know about the existence and construction of fixpoints. For any monotone function $f \colon \mathcal{P}(A) \to \mathcal{P}(A)$, f has fixpoints, and the least/greatest fixpoints are given as follows:

$$\mathbf{lfp}\, f = \bigcap\{A_0 \subseteq A \mid f(A_0) \subseteq A_0\} \qquad\qquad \mathbf{gfp}\, f = \bigcup\{A_0 \subseteq A \mid A_0 \subseteq f(A_0)\}$$

Let us look at the $\mathbf{lfp}\, f$ as the discussion for $\mathbf{gfp}\, f$ is similar. From the construction above, we know two things about $\mathbf{lfp}\, f$. Firstly, it is a fixpoint, so $\mathbf{lfp}\, f = f(\mathbf{lfp}\, f)$. Secondly, for every A_0 such that $f(A_0) \subseteq A_0$ (i.e., A_0 is a pre-fixpoint of f), $\mathbf{lfp}\, f \subseteq A_0$.

Our goal is to incorporate Theorem 2.1 into matching logic so we can obtain a unifying foundation for specifying and reasoning fixpoints that can handle all instances and examples of fixpoints, including inductive datatypes, temporal operators, reachability rules, and so on. Luckily, matching logic patterns fit nicely with the setting of Theorem 2.1 because for any pattern φ and a free variable x in it, we can regard φ (w.r.t. x) as a function over the powerset domain in the following sense: $\psi \mapsto \varphi[\psi/x]$ for every ψ. Here, ψ is a pattern (so semantically, a set) and $\varphi[\psi/x]$ is the result of "applying" φ to ψ. If φ is positive in x, then the corresponding function $\psi \mapsto \varphi[\psi/x]$ is monotone and thus has fixpoints. We denote the least and the greatest fixpoints as $\mu x \,.\, \varphi$ and $\nu x \,.\, \varphi$, respectively. We also know two things about $\mu x \,.\, \varphi$ (and similarly for $\nu X \,.\, \varphi$). Firstly, it is a fixpoint, so $\vdash (\mu x \,.\, \varphi) \leftrightarrow \varphi[(\mu x \,.\, \varphi)/x]$. Secondly, it is smaller than any pre-fixpoint, so $\vdash \varphi[\psi/x] \to \psi$ implies $\vdash (\mu x \,.\, \varphi) \to \psi$. Similarly for $\nu x \,.\, \varphi$, we know that it is a fixpoint, so $\vdash (\nu x \,.\, \varphi) \leftrightarrow \varphi[(\nu x \,.\, \varphi)/x]$. Secondly, it is larger than any post-fixpoint, so $\vdash \psi \to \varphi[\psi/x]$ implies $\vdash \psi \to \nu x \,.\, \varphi$.

If we could define $\mu x \,.\, \varphi$ and $\nu x \,.\, \varphi$ as notation in matching logic, just like how $\forall x \,.\, \varphi \equiv \neg \exists x \,.\, \neg \varphi$ is defined, we would have a directly logical incarnation of Theorem 2.1 in matching logic and obtain a unifying logical foundation to specify and reason about any types of fixpoints. Unfortunately, it turns out that $\mu x \,.\, \varphi$ and $\nu x \,.\, \varphi$ cannot be defined as notation in matching logic. We have to extend matching logic with a new set of *set variables*, denoted by X, Y, etc., and introduce an explicit μ operator to build least fixpoints. Greatest fixpoints can be built by the dual ν operator. We call the extended logic matching μ-logic to emphasize that it has an explicit μ operator. The purpose of this chapter is to formally present matching μ-logic and introduce its extended syntax, semantics, and proof rules.

4.1 NECESSITY OF EXTENSION

We explain why matching logic must be extended to support fixpoints. Let us assume that we find a way to define fixpoints in matching logic as notation. That means that $\mu x \,.\, \varphi$ is a matching logic pattern, where x is a regular variable of matching logic. Then we show that the following rule fails to hold:

$$\textsc{(Equivalence Congruence)} \quad \frac{\varphi_1 \leftrightarrow \varphi_2}{(\mu x \,.\, \varphi_1) \leftrightarrow (\mu x \,.\, \varphi_2)}$$

Note that (Equivalence Congruence) is a highly desired property that is expected to hold in any reasonable formal system. Its failure is thus strong evidence that $\mu x \,.\, \varphi$ must not, or at least, should not be defined as notation. It is more natural and reasonable to extend matching logic to matching μ-logic in order to support fixpoints.

To see why (Equivalence Congruence) fails, let us first note that $\mu x \,.\, x$ should be equivalent to $\bot$. This is because $\mu x \,.\, x$ represents the identity function, whose least fixpoint is the empty set, i.e., $\bot$. Let us also note that $\mu x \,.\, c$ should be equivalent to c for a constant symbol c. This is because $\mu x \,.\, c$ represents a constant function that returns c for all inputs. So its only fixpoint is c itself.

We now build a counterexample of (Equivalence Congruence). We define a matching logic theory $\Gamma = \{x, c\}$. The axiom x enforces the underlying carrier set to be a singleton set, say $\{\star\}$. The axiom c enforces the interpretation of c to be $\{\star\}$, too. Thus we have $\Gamma \vDash x \leftrightarrow c$. However, $\Gamma \nvDash (\mu x \,.\, x) \leftrightarrow (\mu x \,.\, c)$, because the left-hand side should be equivalent to $\bot$ while the right-hand side should be equivalent to c, and $\Gamma \nvDash \bot \leftrightarrow c$. Thus, (Equivalence Congruence) fails to hold.

It means that if we were to define fixpoints in matching logic as notation, we either need to drop the highly desired property (Equivalence Congruence) or live in a weird world where $\mu x \,.\, x$ is not equivalent to $\bot$ (or $\mu x \,.\, c$ is not equivalent to c). We want neither of the above. Thus, we conclude that the proper way to add fixpoint support to in matching logic is to extend it to matching μ-logic, which we present in Section 4.2.

As a side remark, (Equivalence Congruence) does hold in matching μ-logic; see Lemma 4.4.

4.2 MATCHING μ-LOGIC SYNTAX, SEMANTICS, AND PROOF SYSTEM

We define matching μ-logic syntax, semantics, and proof system and present some basic results about its formal reasoning.

4.2.1 Matching μ-logic syntax and semantics

Definition 4.1. A *matching μ-logic signature* or simply a *signature* (S, Σ) is the same as a matching logic signature in Definition 2.45. Given a matching μ-logic signature (S, Σ), an S-indexed set $EV = \{EV_s\}_{s \in S}$ of *element variables* denoted by $x : s$, $y : s$, etc., and an S-indexed set $SV = \{SV_s\}_{s \in S}$ of *set variables* denoted by $X : s$, $Y : s$, etc., the syntax of *matching μ-logic patterns* or simply *patterns* is given by extending the syntax of matching logic with the following grammar rules:

$$\begin{array}{lll} \underline{\text{matching } \mu\text{-logic patterns}} & \varphi_s ::= & \text{(syntax of matching logic)} \\ & & \mid X : s \\ & & \mid \mu X : s \,.\, \varphi_s \end{array}$$

where $\mu X : s \,.\, \varphi_s$ requires that φ_s is *positive* in $X : s$, i.e.,, $X : s$ does not occur in an odd number of $\neg$'s.

We feel free to drop the sorts when they are understood or not important. The notion of free variables $\mathit{freeVar}(\varphi)$ and capture-avoiding substitution $\varphi[\psi/x]$ and $\varphi[\psi/X]$ are defined in the usual way. We define $\nu X \,.\, \varphi$ as follows

$$\nu X \,.\, \varphi \equiv \neg \mu X \,.\, \neg \varphi[\neg X/X]$$

Note that there are three $\neg$'s, not two. Also note that $\neg \varphi[\neg X/X]$ is positive in X whenever φ is positive in X.

Definition 4.2. Given a signature (S, Σ), a *matching μ-logic (S, Σ)-model* or simply *(S, Σ)-model* is the same as a matching logic model in Definition 2.46. Given an (S, Σ)-model $M = (\{M_s\}_{s \in S}, \{\sigma_M\}_{\sigma \in \Sigma})$, a *matching μ-logic M-valuation* or simply *M-valuation* is a pair $\rho = (\rho_{EV}, \rho_{SV})$ where $\rho_{EV} \colon EV \to M$ is a matching logic M-valuation and $\rho_{SV} \colon SV \to \mathcal{P}(M)$.

Definition 4.3. Given a signature (S, Σ) and an (S, Σ)-model $M = (\{M_s\}_{s \in S}, \{\sigma_M\}_{\sigma \in \Sigma})$, the matching μ-logic interpretation function $|\varphi|_{M,\rho}$ is inductively defined for all φ and ρ as follows:

1. $|X : s|_{M,\rho} = \rho_{SV}(X : s)$ for $X : s \in SV$;
2. $|\mu X : s \,.\, \varphi_s|_{M,\rho} = \mathbf{lfp}(A \mapsto |\varphi|_{M,\rho[A/X : s]})$;
3. The rest rules are the same as Definition 2.47.

Here, $\mathbf{lfp}(A \mapsto |\varphi|_{M,\rho[A/X : s]})$ is the least fixpoint of the function that maps A to $|\varphi|_{M,\rho[A/X : s]}$ for $A \subseteq M$.

The derived semantics of $\mu X : s \,.\, \varphi_s$ and $\nu X : s \,.\, \varphi_s$ are as follows:

$$|\mu X : s \,.\, \varphi_s|_{M,\rho} = \mathbf{lfp}(A \mapsto |\varphi|_{M,\rho[A/X : s]}) = \bigcap\{A \subseteq M_s \mid |\varphi|_{M,\rho[A/X : s]} \subseteq A\}$$
$$|\nu X : s \,.\, \varphi_s|_{M,\rho} = \mathbf{gfp}(A \mapsto |\varphi|_{M,\rho[A/X : s]}) = \bigcup\{A \subseteq M_s \mid A \subseteq |\varphi|_{M,\rho[A/X : s]}\}$$

A *matching μ-logic theory* or simply *theory* Γ is a set of patterns/axioms. We define $M \vDash \varphi$, $M \vDash \Gamma$, and $\Gamma \vDash \varphi$ in the usual way. Note that if $\psi \in \Gamma$ has free set variables, then those set variables are effectively universally quantified. This way, matching μ-logic allows to write axioms that features monadic universal second-order quantification at the top of axioms. This fact is useful for defining powersets (Section 5.6.1) and SOL (Section 5.6) as matching μ-logic theories.

4.2.2 Matching μ-logic proof system $\mathcal{H}_\mu$

The matching μ-logic proof system $\mathcal{H}_\mu$, as shown in Figure 4.1, extends the matching logic proof system $\mathcal{H}$ in Figure 3.2 with three proof rules: (Substitution), (Pre-Fixpoint), and (Knaster Tarski). The latter two have been discussed in Section 4.1. They are a direct logical incarnation of the Knaster-Tarski fixpoint theorem (Theorem 2.1) into matching μ-logic. The (Substitution) rule captures the semantic effect that a free set variable in a matching μ-logic axiom is universally quantified. So if φ is provable and $X \in \mathit{freeVar}(\varphi)$, then $\varphi[\psi/X]$ is still provable for any ψ. We use $\vdash_{\mathcal{H}_\mu}$ or simply $\vdash$ to denote the corresponding

(System $\mathcal{H}$)	all proof rules of $\mathcal{H}$ in Figure 3.2
(Substitution)	$\dfrac{\varphi}{\varphi[\psi/X]}$
(Pre-Fixpoint)	$\varphi[\mu X\,.\,\varphi/X] \to \mu X\,.\,\varphi$
(Knaster Tarski)	$\dfrac{\varphi[\psi/X] \to \psi}{\mu X\,.\,\varphi \to \psi}$

Figure 4.1: Matching μ-Logic Proof System $\mathcal{H}_\mu$

provability relation of $\mathcal{H}_\mu$. Since $\mathcal{H}_\mu$ extends $\mathcal{H}$, we have that $\Gamma \vdash_{\mathcal{H}} \varphi$ implies $\Gamma \vdash \varphi$ for any matching logic theory Γ and pattern φ.

Theorem 4.1. $\mathcal{H}_\mu$ is sound, i.e., for any Γ and φ, $\Gamma \vdash \varphi$ implies $\Gamma \vDash \varphi$.

Proof. We only need to verify the soundness of (Substitution), (Pre-Fixpoint), and (Knaster Tarski).

For (Substitution), let us assume any $M \vDash \varphi$. By definition, $|\varphi|_{M,\rho} = M$ for all ρ. Our goal is to show $M \vDash \varphi[\psi/X]$. Let ρ be any valuation. We have $|\varphi[\psi/X]|_{M,\rho} = |\varphi|_{M,\rho[|\psi|_{M,\rho}/X]}$. Note that $\rho[|\psi|_{M,\rho}/X]$ is just another valuation, so $|\varphi|_{M,\rho[|\psi|_{M,\rho}/X]} = M$ by assumption.

For (Pre-Fixpoint), let ρ be any valuation. Our goal is to prove $|\varphi[\mu X\,.\,\varphi/X] \to \mu X\,.\,\varphi|_{M,\rho} = M$. By definition, $|\varphi[\mu X\,.\,\varphi/X]|_{M,\rho} = |\varphi|_{M,\rho[|\mu X\,.\,\varphi|_{M,\rho}/X]}$, and $|\mu X\,.\,\varphi|_{M,\rho} = \bigcap\{A \mid |\varphi|_{M,\rho\rho[A/X]} \subseteq A\}$. By Theorem 2.1, $|\mu X\,.\,\varphi|_{M,\rho}$ itself is a fixpoint of $|\varphi|_{M,\rho[A/X]} = A$. Therefore, $|\varphi|_{M,\rho[|\mu X\,.\,\varphi|_{M,\rho}/X]} = |\mu X\,.\,\varphi|_{M,\rho}$.

For (Knaster Tarski), let $M \vDash \varphi[\psi/X] \to \psi$. Our goal is to prove $M \vDash \mu X\,.\,\varphi \to \psi$. Let ρ be any valuation. We need to prove $|\mu X\,.\,\varphi|_{M,\rho} \subseteq |\psi|_{M,\rho}$. Note that $|\mu X\,.\,\varphi|_{M,\rho}$ is defined as the least fixpoint of $|\varphi|_{M,\rho[A/X]} = A$. By Theorem 2.1, it suffices to prove $|\psi|_{M,\rho}$ is a pre-fixpoint, i.e., $|\varphi|_{M,\rho[|\psi|_{M,\rho}/X]} \subseteq |\psi|_{M,\rho}$. This is given by our assumption, $M \vDash \varphi[\psi/X] \to \psi$. This implies that $|\varphi[\psi/X]|_{M,\rho} \subseteq |\psi|_{M,\rho}$, i.e., $|\varphi|_{M,\rho[|\psi|_{M,\rho}/X]} \subseteq |\psi|_{M,\rho}$. QED.

4.2.3 Basic properties about $\mathcal{H}_\mu$

We prove some basic properties about $\mathcal{H}_\mu$. Firstly, we generalized Lemma 3.1 matching μ-logic with set variables and the μ operator.

Lemma 4.1. $|\varphi[\psi/X]|_{M,\rho} = |\varphi|_{M,\rho[\rho(\psi))/X]}$ for all $X \in SV$.

Proof. The proof is the same as Lemma 3.1. The only interesting case is when $\varphi \equiv \mu Z \,.\, \varphi_1$. By α-renaming, we can safely assume $Z \notin \mathit{freeVar}(\psi)$. We have:

$$\begin{aligned}
|(\mu Z \,.\, \varphi_1)[\psi/X]|_{M,\rho} &= |\mu Z \,.\, (\varphi_1[\psi/X])|_{M,\rho} \\
&= \bigcap\{A \mid |\varphi_1[\psi/X]|_{M,\rho[A/Z]} \subseteq A\} \\
&= \bigcap\{A \mid |\varphi_1|_{M,\rho[A/Z][|\psi|_{M,\rho[A/Z]}/X]} \subseteq A\} \\
&= \bigcap\{A \mid |\varphi_1|_{M,\rho[A/Z][|\psi|_{M,\rho}/X]} \subseteq A\} \\
&= \bigcap\{A \mid |\varphi_1|_{M,\rho[|\psi|_{M,\rho}/X][A/Z]} \subseteq A\} \\
&= |\mu Z \,.\, \varphi_1|_{M,\rho[|\psi|_{M,\rho}/X]} \\
&= |\varphi|_{M,\rho[|\psi|_{M,\rho}/X]}.
\end{aligned}$$

QED.

Then we show that $\mu X \,.\, \varphi$ is indeed a fixpoint in $\mathcal{H}_\mu$.

Lemma 4.2. $\vdash \mu X \,.\, \varphi \leftrightarrow \varphi[\mu X \,.\, \varphi/X]$.

Proof. We prove both directions.

(Case "$\to$"). Apply (Knaster Tarski), and we prove $\vdash \varphi[(\varphi[\mu X \,.\, \varphi/X])/X] \to \varphi[\mu X \,.\, \varphi/X]$. By Lemma 4.6, and the fact that φ is positive in X, we just need to prove $\vdash \varphi[\mu X \,.\, \varphi/X] \to \varphi$, which is proved by (Pre-Fixpoint).

(Case "$\leftarrow$") is exactly (Pre-Fixpoint). QED.

The proof system $\mathcal{H}_\mu$ only defines (Pre-Fixpoint) and (Knaster Tarski) for $\mu X \,.\, \varphi$. Here we show that their dual versions also hold for $\nu X \,.\, \varphi$.

Lemma 4.3. The following propositions hold:

- (Pre-Fixpoint): $\vdash \nu X \,.\, \varphi \to \varphi[\nu X \,.\, \varphi/X]$;
- (Knaster Tarski): $\vdash \psi \to \varphi[\psi/X]$ implies $\vdash \psi \to \nu X \,.\, \varphi$.

Proof. Simply unfold $\nu X \,.\, \varphi$ to $\neg \mu X \,.\, \neg(\varphi[\neg X/X])$ and use the version of (Pre-Fixpoint) and (Knaster Tarski) for the least fixpoints. QED.

We verify that (Equivalence Congruence) in Section 4.1 is indeed derivable in $\mathcal{H}_\mu$.

Lemma 4.4. $\Gamma \vdash \varphi_1 \to \varphi_2$ implies $\Gamma \vdash \mu X \,.\, \varphi_1 \to \mu X \,.\, \varphi_2$.

Proof. Use (Knaster Tarski), and then (Substitution). QED.

Lemma 4.5. For any context C, we have $\Gamma \vdash \varphi_1 \leftrightarrow \varphi_2$ iff $\Gamma \vdash C[\varphi_1] \leftrightarrow C[\varphi_2]$.

Proof. Carry out induction on the structure of C. Except the case $C \equiv \mu X \,.\, C_1$, all other cases have been proved in Proposition 3.5. While the μ-case is proved by Lemma 4.4. QED.

Lemma 4.5 allows us to in-place unfold a fixpoint pattern in any context. That is, we can freely replace $\mu X \,.\, \varphi$ (or $\nu X \,.\, \varphi$) with $\varphi[(\mu X \,.\, \varphi)/X]$ (or $\varphi[(\nu X \,.\, \varphi)/X]$) in any context.

Lemma 4.6. A context C is positive if it is positive in $\square$; otherwise, it is negative. Let $\Gamma \vdash \varphi_1 \to \varphi_2$. We have

$$\begin{aligned} &\Gamma \vdash C[\varphi_1] \to C[\varphi_2] && \text{if } C \text{ is positive,} \\ &\Gamma \vdash C[\varphi_2] \to C[\varphi_1] && \text{if } C \text{ is negative.} \end{aligned}$$

Proof. Carry out induction on the structure of C. The cases when C is a propositional/FOL context are trivial. The case when C is a symbol application is proved by (Framing). The case when C is μ is proved by Lemma 4.4. QED.

Lemma 4.7. Let ψ be a predicate pattern, i.e., $\vdash \psi = \top \vee \psi = \bot$, and C be a context where $\square$ is not under any μ's. We have $\vdash \psi \wedge C[\varphi] \leftrightarrow \psi \wedge C[\psi \wedge \varphi]$ for all φ.

Proof. Carry out induction on the structure of C. The cases when C is a propositional/FOL context are trivial. The case when C is a symbol application is proved using the fact that predicate patterns propagate through symbols. Since $\square$ does not occur under any μ's, we have considered all the cases. QED.

Lemma 4.8. Let ψ be a predicate pattern and φ be a pattern. Let X be a set variable that does not occur under any μ's in φ, and $X \notin \mathit{freeVar}(\psi)$. We have $\vdash \psi \wedge \mu X \,.\, \varphi \leftrightarrow \mu X \,.\, (\psi \wedge \varphi)$.

Proof. Note that "$\leftarrow$" is proved by Lemma 4.4. We only need to prove "$\to$". By propositional reasoning, the goal becomes $\vdash \mu X \,.\, \varphi \to \psi \to \mu X \,.\, (\psi \wedge \varphi)$ and we apply (Knaster Tarski). We obtain $\vdash \psi \wedge \varphi[\psi \to \mu X \,.\, (\psi \wedge \varphi)/X] \to \mu X \,.\, (\psi \wedge \varphi)$. By (Pre-Fixpoint), we just need to prove $\vdash \psi \wedge \varphi[\psi \to \mu X \,.\, (\psi \wedge \varphi)/X] \to \psi \wedge \varphi[\mu X \,.\, (\psi \wedge \varphi)/X]$. By Lemma 4.8, we just need to prove $\vdash \psi \wedge \varphi[\psi \wedge (\psi \to \mu X \,.\, (\psi \wedge \varphi))/X] \to \psi \wedge \varphi[\mu X \,.\, (\psi \wedge \varphi)/X]$, which then by Lemma 4.6 becomes $\vdash \psi \wedge \varphi[\psi \wedge (\mu X \,.\, (\psi \wedge \varphi))/X] \to \psi \wedge \varphi[\mu X \,.\, (\psi \wedge \varphi)/X]$, which then follows by Lemma 4.8. QED.

We present a deduction theorem for $\mathcal{H}_\mu$.

Theorem 4.2. Let Γ be a theory that includes the definedness symbols and axioms, and φ, ψ be two patterns. If $\Gamma \cup \{\psi\} \vdash \varphi$ and the proof (1) does not use (Universal Generalization) on free element variables in ψ; (2) does not use (Knaster Tarski), *unless* (Knaster Tarski) set variable X does not occur under any μ's in φ and $X \notin \mathit{freeVar}(\psi)$; (3) does not use (Substitution) on free set variables in ψ, then $\Gamma \vdash \lfloor \psi \rfloor \to \varphi$.

Proof. Carry out induction on the length of the proof $\Gamma \cup \{\psi\} \vdash \varphi$. (Base Case) and (Induction Case) for (Modus Ponens) and (Universal Generalization) are proved as in Theorem 4.2. We only need to prove (Induction Case) for (Knaster Tarski) and (Substitution).

(Knaster-Tarski). Suppose $\varphi \equiv \mu X \,.\, \varphi_1 \to \varphi_2$. We should prove that $\Gamma \vdash \lfloor \psi \rfloor \to (\mu X \,.\, \varphi_1 \to \varphi_2)$, i.e., $\Gamma \vdash \lfloor \psi \rfloor \wedge \mu X \,.\, \varphi_1 \to \varphi_2$. Note that $\lfloor \psi \rfloor$ is a predicate pattern. By Lemma 4.8, our goal becomes $\Gamma \vdash \mu X \,.\, (\lfloor \psi \rfloor \wedge \varphi_1) \to \varphi_2$. By (Knaster Tarski), we need to prove $\Gamma \vdash (\lfloor \psi \rfloor \wedge \varphi_1)[\varphi_2/X] \to \varphi_2$. Note that $X \notin \mathit{freeVar}(\lfloor \psi \rfloor)$, so the above becomes $\Gamma \vdash \lfloor \psi \rfloor \wedge \varphi_1[\varphi_2/X] \to \varphi_2$, i.e., $\Gamma \vdash \lfloor \psi \rfloor \to \varphi_1[\varphi_2/X] \to \varphi_2$, which is our induction hypothesis.

(Substitution). Trivial. Note that $X \notin \mathit{freeVar}(\psi)$. QED.

4.3 REDUCTION TO MONADIC SECOND-ORDER LOGIC

It is shown in [2] that matching logic patterns can be translated into equivalent formulas in pure predicate logic with equality (i.e., FOL that is extended with equality and has no function symbols). The idea is to define for every pattern φ a corresponding formula written $\mathit{PL}_2(\varphi, r)$ where r is a fresh variable, with the intuition that r matches φ iff $\mathit{PL}_2(\varphi, r)$ holds. In other words, we reduce the powerset semantics of patterns to the classical FOL semantics by defining the membership relation. This way, a pattern φ is valid (i.e., is matched by every element) iff $\mathit{PL}(\varphi) \equiv \forall r : \mathit{PL}_2(\varphi, r)$ holds.

Matching μ-logic extends matching μ-logic with set variables and the μ operator, and thus goes beyond the expressive power of FOL with equality. However, as we will show later, there is a reduction from matching μ-logic to SOL, or more precisely, to monadic SOL (abbreviated as MSO). The intuition is the same. For any matching μ-logic pattern φ, we define a corresponding MSO formula $\mathit{MSO}_2(\varphi, r)$ with a fresh variable r, such that r matches φ iff $\mathit{MSO}_2(\varphi, r)$ holds. Then, φ is valid iff $\mathit{MSO}(\varphi) \equiv \forall r \,.\, \mathit{MSO}_2(\varphi, r)$ holds.

The reduction from matching μ-logic to MSO is given as follows. Given a matching μ-logic signature (S, Σ) and the two sets $EV = \{EV_s\}_{s \in S}$ and $SV = \{SV_s\}_{s \in S}$ of element and set variables, we define a MSO signature $(S^{\mathit{MSO}}, C^{\mathit{MSO}}, \Pi^{\mathit{MSO}})$ by letting $S^{\mathit{MSO}} = S$,

$C^{MSO} = \emptyset$, and $\Pi^{MSO} = \{\pi_\sigma \colon s_1 \times \cdots \times s_n \times s \mid \sigma \in \Sigma_{s_1 \ldots s_n, s}\}$. All element variables in EV are included as MSO element variables. For every set variable $X : s \in SV_s$, we add it as a unary predicate variable over sort s. We define the translation from (S, Σ)-patterns to $(S^{MSO}, C^{MSO}, \Pi^{MSO})$-formulas as follows:[1]

$$\begin{aligned}
\mathit{MSO}(\varphi) &= \forall r \,.\, \mathit{MSO}_2(\varphi, r) \\
\mathit{MSO}_2(x, r) &= x = r \\
\mathit{MSO}_2(\sigma(\varphi_1, \ldots, \varphi_n), r) &= \exists r_1 \ldots \exists r_n \,.\, \mathit{MSO}_2(\varphi_i, r_i) \wedge \pi_\sigma(r_1, \ldots, r_n, r) \\
\mathit{MSO}_2(\neg\varphi, r) &= \neg \mathit{MSO}_2(\varphi, r) \\
\mathit{MSO}_2(\varphi_1 \wedge \varphi_2, r) &= \mathit{MSO}_2(\varphi_1, r) \wedge \mathit{MSO}_2(\varphi_2, r) \\
\mathit{MSO}_2(\exists x \,.\, \varphi, r) &= \exists x \,.\, \mathit{MSO}_2(\varphi, r) \\
\mathit{MSO}_2(X, r) &= X(r) \\
\mathit{MSO}_2(\mu X \,.\, \varphi, r) &= \forall X \,.\, (\forall r' \,.\, \mathit{MSO}_2(\varphi, r') \to X(r')) \to X(r) \\
\mathit{MSO}(\Gamma) &= \{\mathit{MSO}(\psi) \mid \psi \in \Gamma\}
\end{aligned}$$

As said, $\mathit{MSO}_2(\varphi, r)$ captures the intuition that r matches φ. The top translation $\mathit{MSO}(\varphi)$ captures the intuition that φ is valid iff it is matched by all r. Therefore, we have the following theorem.

Theorem 4.3. For any Γ and φ, $\Gamma \vDash \varphi$ iff $\mathit{MSO}(\Gamma) \vDash_{\mathsf{SOL}} \mathit{MSO}(\varphi)$.

Proof. It suffices to show that there exists a bijection between (S, Σ)-models M and $(S^{MSO}, C^{MSO}, \Pi^{MSO})$-models M' such that $M \vDash \varphi$ iff $M' \vDash_{\mathsf{SOL}} \mathit{MSO}(\varphi)$. The bijection is defined as follows:

1. $M'_s = M_s$ for all $s \in S$;
2. $\pi_{\sigma M'} = \{(a_1, \ldots, a_n, b) \mid b \in \sigma_M(a_1, \ldots, a_n)\}$.

To show that $M \vDash \varphi$ iff $M' \vDash_{\mathsf{SOL}} \mathit{MSO}(\varphi)$, it suffices to show $a \in |\varphi|_{M,\rho}$ iff $M', \rho[a/r] \vDash_{\mathsf{SOL}} \mathit{MSO}_2(\varphi)$, which follows by structural induction on φ. We only need to prove the cases for $\mathit{MSO}_2(X)$ and $\mathit{MSO}_2(\mu X \,.\, \varphi)$ because the other cases are the same as the translation from matching logic to predicate logic in [2, Section 10].

(Case $\mathit{MSO}_2(X, r)$). We have $a \in |X|_{M,\rho}$ iff $a \in \rho(X)$ iff $M', \rho[a/r] \vDash_{\mathsf{SOL}} X(r)$.

(Case $\mathit{MSO}_2(\mu X \,.\, \varphi, r)$). We have $a \in |\mu X \,.\, \varphi|_{M,\rho}$ iff $a \in \mathbf{lfp}\,(A \mapsto |\varphi|_{M,\rho[A/X]})$ iff $a \in \bigcap\{A \mid |\varphi|_{M,\rho[A/X]} \subseteq A\}$ iff for every A, $|\varphi|_{M,\rho[A/X]} \subseteq A$ implies $a \in A$. Note

[1] The unsorted version of the translation of $\mu X \,.\, \varphi$ was initially proposed by Adam Fiedler.

that for any fixed A, $|\varphi|_{M,\rho[A/X]} \subseteq A$ iff for every a, $a \in |\varphi|_{M,\rho[A/X]}$ implies $a \in A$, iff (by the induction hypotheses) $M', \rho[a/r, A/X] \vDash_{\mathsf{SOL}} \forall r' \,.\, \mathit{MSO}_2(\varphi, r') \to X(r')$. Therefore, we have that "for every A, $|\varphi|_{M,\rho[A/X]} \subseteq A$ implies $a \in A$" is equivalent to "for every A, $M', \rho[a/r, A/X] \vDash_{\mathsf{SOL}} \forall r' \,.\, \mathit{MSO}_2(\varphi, r') \to X(r')$". The latter is equivalent to $M', \rho[a/r] \vDash_{\mathsf{SOL}} \forall X \,.\, (\forall r' \,.\, \mathit{MSO}_2(\varphi, r') \to X(r')) \to X(r)$. QED.

As a closing remark, we point out that translating patterns to MSO introduces new complexity to not only specifying properties but also reasoning about them. Such complexity comes from the fact that during the translation new quantifiers are introduced, such as in $\mathit{MSO}_2(\sigma(\varphi_1, \ldots, \varphi), r)$ and $\mathit{MSO}_2(\mu X \,.\, \varphi, r)$. Therefore, it is more difficult to reason about the MSO translations than to directly reason about matching μ-logic patterns using $\mathcal{H}_\mu$.

Chapter 5: EXPRESSIVE POWER

In this chapter we study the expressive power of matching μ-logic. We will consider various logics, calculi, and foundations of computation and show how to define them in matching μ-logic as theories.

5.1 DEFINING RECURSIVE SYMBOLS

We know that in matching μ-logic, we can use $\mu X \,.\, \varphi$ to specify a recursive set that satisfies the equation $X = \varphi$, where the interesting case is when X has free occurrences in φ. For example, $\mu X \,.\, 3 \vee \mathit{plus}(X, X)$ specifies the set of all nonzero multiples of 3, which is the smallest set that includes 3 and is closed under *plus*. Intuitively, $\mu X \,.\, 3 \vee \mathit{plus}(X, X)$ defines a constant symbol $\mathsf{m3} \in \Sigma_{\lambda,\mathsf{Nat}}$ by the following recursive definition:

$$\mathsf{m3} \in \Sigma_{\lambda,\mathsf{Nat}} \qquad\qquad \mathsf{m3} =_{\mathbf{lfp}} 3 \vee \mathit{plus}(\mathsf{m3}, \mathsf{m3}).$$

Our goal is to generalize the above and define recursive symbols of any arities. For example, we would like to define a unary symbol *collatz* $\in \Sigma_{\mathsf{Nat},\mathsf{Nat}}$ by the following recursive definition:

$$\mathit{collatz}(n) =_{\mathbf{lfp}} n \vee (\mathit{even}(n) \wedge \mathit{collatz}(n/2)) \vee (\mathit{odd}(n) \wedge \mathit{collatz}(3n+1))$$

Intuitively, $\mathit{collatz}(n)$ captures the set of all numbers in the Collatz sequence starting from n, where a Collatz sequence is obtained by repeating the following procedure: if the current number is even then the next number is $n/2$; otherwise, the next number is $3n + 1$. However, the μ operator in matching μ-logic can only be applied to set variables, not symbols, so the following attempt is syntactically wrong:

$$\mathit{collatz}(n) = \mu\, \sigma(n) \,.\, n \vee (\mathit{even}(n) \wedge \sigma(n/2)) \vee (\mathit{odd}(n) \wedge \sigma(3n+1))$$

One possible solution is to extend matching μ-logic with recursive symbols and allow μ to bind symbol variables, and not just set variables. We need to extend the syntax, semantics, and proof system accordingly, similarly to how LFP extends FOL with predicate variables. The other approach, which will be presented in this section, is to define recursive symbols using axioms. After all, the proof rules (Pre-Fixpoint) and (Knaster Tarski) in Figure 4.1 are a direct incarnation of the Knaster-Tarski fixpoint theorem (Theorem 2.1). The latter has been repeatedly demonstrated to serve as a solid if not the main foundation for recursion.

Therefore, matching μ-logic should be sufficient in practice for defining one's desired approach to recursion/induction/fixpoints as theories, just like how equality, membership, and functions are defined as theories, as shown in Section 2.13.2.

Our definition of recursive symbols is based on the principle of currying-uncurrying [38, 39], which is is used in various settings (e.g., simply-typed lambda calculus [40]) as a means to reduce the study of multiary functions to unary functions. The principle of currying-uncurrying gives us a one-to-one correspondence between an n-ary symbol $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ with a set variable $\sigma : s_1 \otimes \cdots \otimes s_n \otimes s$. Here $s_1 \otimes \cdots \otimes s_n \otimes s$ is a sort whose carrier set is axiomatically defined to be the product of the carrier sets of $s_1, \ldots, s_n, s$, i.e., $M_{s_1 \otimes \cdots \otimes s_n \otimes s} = M_{s_1} \times \cdots \times M_{s_n} \times M_s$. Then, any recursive symbol from $s_1, \ldots, s_n$ to s can be defined using the μ operator and the set variable $\sigma : s_1 \otimes \cdots \otimes s_n \otimes s$.

Definition 5.1. For sets $M_{s_1}, \ldots, M_{s_n}, M_s$, the *principle of currying-uncurrying* means the following isomorphism:

$$\mathcal{P}(M_{s_1} \times \cdots \times M_{s_n} \times M_s) \underset{\mathit{uncurry}}{\overset{\mathit{curry}}{\rightleftarrows}} [M_{s_1} \times \cdots \times M_{s_n} \to \mathcal{P}(M_s)]$$

given by

$$\begin{aligned} \mathit{curry}(\alpha)(a_1, \ldots, a_n) &= \{b \in M_s \mid (a_1, \ldots, a_n, b) \in \alpha\} \\ \mathit{uncurry}(f) &= \{(a_1, \ldots, a_n, b) \mid b \in f(a_1, \ldots, a_n)\}. \end{aligned}$$

for all $\alpha \subseteq M_{s_1} \times \cdots \times M_{s_n} \times M_s$, $a_i \in M_{s_i}$, $1 \leq i \leq n$, and $f \colon M_{s_1} \times \cdots \times M_{s_n} \to \mathcal{P}(M_s)$.

To use the principle of currying-uncurrying in matching μ-logic, we first need to define product sets as theories.

Definition 5.2. For sorts $s_1, \ldots, s_n$, we define the *product sort* $s_1 \otimes \cdots \otimes s_n$ with a symbol set $\Sigma^{\mathsf{product}}$ and an axiom set $\Gamma^{\mathsf{product}}$, where

$$\Sigma^{\mathsf{product}} = \{\langle \_, \ldots, \_ \rangle \in \Sigma^{\mathsf{product}}_{s_1 \ldots s_n, s_1 \otimes \cdots \otimes s_n}\} \cup \{\mathsf{proj}_i \in \Sigma^{\mathsf{product}}_{s_1 \otimes \cdots \otimes s_n, s_i} \mid 1 \leq i \leq n\}$$

and $\Gamma^{\mathsf{product}}$ includes the following axioms:

(FUNCTION) $\langle \_, \ldots, \_ \rangle \colon s_1 \times \cdots \times s_n \to s_1 \otimes \cdots \otimes s_n$

(FUNCTION) $\mathsf{proj}_i \colon s_1 \otimes \cdots \otimes s_n \to s_i$ with $1 \leq i \leq n$

(INJECTIVITY) $\langle x_1, \ldots, x_n \rangle = \langle y_1, \ldots, y_n \rangle = x_1 = y_1 \wedge \cdots \wedge x_n = y_n$

(PROJECTION) $\mathsf{proj}_i(\langle x_1, \ldots, x_n \rangle) = x_i$ with $1 \leq i \leq n$

$$\text{(PRODUCT)} \qquad \exists x_1 \ldots \exists x_n \,.\, \langle x_1, \ldots, x_n \rangle$$

Proposition 5.1. *For $M \vDash \Gamma^{\mathsf{product}}$, there is an isomorphism $M_{s_1 \otimes \cdots \otimes s_n} \underset{j}{\overset{i}{\rightleftharpoons}} M_{s_1} \times \cdots \times M_{s_n}$.*

Proof. For $a_i \in M_{s_i}$, $1 \le i \le n$, we define $\langle a_1, \ldots, a_n \rangle_M$ by

$$\{\langle a_1, \ldots, a_n \rangle_M\} = (\langle \_, \ldots, \_ \rangle)_M(a_1, \ldots, a_n)$$

where $(\langle \_, \ldots, \_ \rangle)_M \colon M_{s_1} \times \cdots \times M_{s_n} \to \mathcal{P}(M_{s_1 \otimes \cdots \otimes s_n})$ is the interpretation of $\langle \_, \ldots, \_ \rangle$ in M. This is well-defined because of (FUNCTION), which states that $\langle \_, \ldots, \_ \rangle$ is a function and returns thus only singleton sets. Let $j \colon M_{s_1} \times \cdots \times M_{s_n} \to M_{s_1 \otimes \cdots \otimes s_n}$ be a function defined by $j(a_1, \ldots, a_n) = \langle a_1, \ldots, a_n \rangle_M$ for all $a_i \in M_{s_n}$, $1 \le i \le n$. Then j is surjective because of (PRODUCT). Furthermore, j is injective because of (INJECTIVITY). Therefore, j is bijective and has an inverse i, given by $i(\langle a_1, \ldots, a_n \rangle_M) = (a_1, \ldots, a_n)$, for all $a_i \in M_{s_n}$, $1 \le i \le n$. Thanks to this isomorphism, we feel free to write $\langle a_1, \ldots, a_n \rangle_M$ just as $(a_1, \ldots, a_n)$. QED.

To define recursive symbols, we often consider the product sort $s_1 \otimes \cdots \otimes s_n \otimes s$, where $s_1, \ldots, s_n$ are the argument sorts and s is the return sort. It is often convenient to add a new *application symbol* $\_(\_, \ldots, \_) \in \Sigma^{\mathsf{product}}_{(s_1 \otimes \cdots \otimes s_n \otimes s)\, s_1 \ldots s_n, s}$ and include the following axioms in $\Gamma^{\mathsf{product}}$:

$$\text{(FUNCTION)} \qquad \_(\_, \ldots, \_) \colon (s_1 \otimes \cdots \otimes s_n \otimes s) \times s_1 \times \cdots \times s_n \to s$$

$$\text{(APPLICATION)} \qquad p(x_1, \ldots, x_n) = \exists y \,.\, y \wedge \langle x_1, \ldots, x_n, y \rangle \in p$$

Intuitively, (APPLICATION) states that $p(x_1, \ldots, x_n)$ includes all y's such that $\langle x_1, \ldots, x_n, y \rangle$ matches p. By tacitly using the same syntax $\_(\_, \ldots, \_)$ for the application symbol given above and the application operator in the matching μ-logic syntax, we blur their distinction. In particular, if $\sigma : s_1 \otimes \cdots \otimes s_n \otimes s$ is a set variable and $\varphi_1, \ldots, \varphi_n$ have sorts $s_1, \ldots, s_n$, respectively, then $\sigma(\varphi_1, \ldots, \varphi_n)$ is a well-formed pattern of sort s.

Now we are ready to define recursive symbols as recursive sets, which are definable using the μ operator.

Definition 5.3. For a symbol $\sigma \in \Sigma_{s_1 \ldots s_1, s}$, we write $\sigma(x_1, \ldots, x_n) =_{\mathsf{lfp}} \varphi$ to mean the following axiom:

$$\sigma(x_1, \ldots, x_n) = (\mu\sigma : s_1 \otimes \cdots \otimes s_n \otimes s \,.\, \exists x_1 \ldots \exists x_n \,.\, \langle x_1, \ldots, x_n, \varphi \rangle)(x_1, \ldots, x_n)$$

where the symbol $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ in φ is tacitly regarded as the set variable $\sigma : s_1 \otimes \cdots \otimes s_n \otimes s$, which is then bound by μ. We call $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ a *recursive symbol* and $\sigma(x_1, \ldots, x_n) =_{\mathbf{lfp}} \varphi$ its *recursive definition.*

Recursive symbols can be used to define various inductive data structures and relations. For example, in Sections 5.2 and 5.3, we show how to define LFP formulas and SL recursive symbols using matching μ-logic recursive symbols. As for formal reasoning, Definition 5.3 is not the most convenient because it involves a lot of detail related to the construction of the product sort. To reason about recursive symbols more easily, we generalize the (Pre-Fixpoint) and (Knaster Tarski) proof rules and prove that they are derivable in matching μ-logic, so we can reason about recursive symbols in the same way as the basic least fixpoints $\mu X \,.\, \varphi$.

Theorem 5.2. Let Γ be a theory with a recursive symbol $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ defined by

$$\sigma(x_1, \ldots, x_n) =_{\mathbf{lfp}} \varphi$$

For any ψ such that

$$\begin{aligned} \Gamma \vdash (&\exists z_1 \ldots \exists z_n \,.\, z_1 \in \varphi_1 \wedge \cdots \wedge z_n \in \varphi_n \wedge \psi[z_1/x_1, \ldots, z_n/x_n]) \\ &\to \psi[\varphi_1/x_1, \ldots, \varphi_n/x_n] \end{aligned} \tag{5.1}$$

for all $\varphi_1, \ldots, \varphi_n$, the following hold:

- (Pre-Fixpoint): $\Gamma \vdash \varphi \to \sigma(x_1, \ldots, x_n)$;
- (Knaster Tarski): $\Gamma \vdash \varphi[\psi/\sigma] \to \psi$ implies $\Gamma \vdash \sigma(x_1, \ldots, x_n) \to \psi$, where $\varphi[\psi/\sigma]$ is the result of substituting all sub-patterns of the form $\sigma(\varphi_1, \ldots, \varphi_n)$ in φ for $\psi[\varphi_1/x_1, \ldots, \varphi_n/x_n]$.

Proof. See Section 5.14.1 QED.

5.2 DEFINING FOL WITH LEAST FIXPOINTS

Recall that FOL with least fixpoint (abbreviated LFP) extends FOL with predicate variables in $PV = \{PV_{s_1 \ldots s_n}\}_{s_1, \ldots, s_n \in S}$ and the following grammar rules (see Definition 2.8):

$$\begin{array}{lll} \underline{\text{LFP formulas}} \quad \varphi ::= (\text{syntax of FOL formulas}) & & \\ \qquad\qquad\qquad\qquad \mid P(t_{s_1}, \ldots, t_{s_n}) & & \text{with } P \in PV_{s_1 \ldots, s_n} \\ \qquad\qquad\qquad\qquad \mid [\mathsf{lfp}_{P, x_1 : s_1 \ldots, x_n : s_n} \varphi](t_{s_1}, \ldots, t_{s_n}) & & \text{with } P \in PV_{s_1 \ldots, s_n} \end{array}$$

We can define LFP in matching μ-logic by extending the theory Γ^{FOL} for FOL in Definition 2.51 with the definitions and notation for recursive symbols in Section 5.1. Furthermore, we add every predicate variable $P \in PV_{s_1 \ldots s_n}$ as a set variable $P \colon s_1 \otimes \cdots \otimes s_n \otimes \mathsf{Formula}$ and define the following notation:

$$\begin{aligned}&[\mathsf{lfp}_{P, x_1 : s_1 \ldots, x_n : s_n} \varphi](t_{s_1}, \ldots, t_{s_n}) \\ &\quad \equiv (\mu P \colon s_1 \otimes \cdots \otimes s_n \otimes \mathsf{Formula} \,.\, \exists x_1 : s_1 \ldots \exists x_n : s_n \,.\, \langle x_1, \ldots, x_n, \varphi \rangle)(t_{s_1}, \ldots, t_{s_n})\end{aligned}$$

Let us use Γ^{LFP} to denote resulting theory. Note that Γ^{LFP} only extends Γ^{FOL} with the generic definitions for recursive symbols and does not include any LFP-specific axioms.

Theorem 5.3. For any LFP formula φ, $\vDash_{\mathsf{LFP}} \varphi$ iff $\Gamma^{\mathsf{LFP}} \vDash \varphi$.

Proof. See Section 5.14.2. QED.

5.3 DEFINING SEPARATION LOGIC WITH RECURSIVE SYMBOLS

Separation logic (abbreviated SL) recursive symbols are a special instance of matching μ-logic recursive symbols. More precisely, SL recursive symbols are matching μ-logic recursive symbols whose return sort is Map—the sort of maps—defined in Definition 2.52. For example, the following recursive definition of singly-linked lists

$$\mathsf{list}(x) =_{\mathbf{lfp}} (x = \mathsf{nil}) \wedge \mathsf{emp} \vee \exists y \,.\, (x \neq \mathsf{nil}) \wedge x \mapsto y * \mathsf{list}(y)$$

is a verbatim definition of a matching μ-logic recursive symbol $\mathsf{list} \in \Sigma^{\mathsf{Map}}_{\mathsf{Nat},\mathsf{Map}}$, without any translation or encoding, thanks to the notation in Section 5.1.

Theorem 5.4. Let M^{Map} be the standard map model in Definition 2.52. For every SL recursive symbol P defined by $P(x_1, \ldots, x_n) =_{\mathbf{lfp}} \psi$, we add $P \in \Sigma^{\mathsf{Map}}_{\mathsf{Nat} \ldots \mathsf{Nat},\mathsf{Map}}$ as a matching μ-logic recursive symbol defined by the same equation $P(x_1, \ldots, x_n) =_{\mathbf{lfp}} \psi$. Let us still use Γ^{SOL} to denote the extended theory. Then for any SL formula φ with recursive symbols, $\vDash_{\mathsf{SOL}} \varphi$ iff $\Gamma^{\mathsf{SOL}} \vDash \varphi$.

Proof. See Section 5.14.3. QED.

5.4 DEFINING EQUATIONAL SPECIFICATIONS

Given that we can define equality and functions in matching logic even without the fixpoint extension (see Section 2.13.2), it is not surprising that we can define equational specifications as theories. Therefore, the point of this section is to formally state and prove an expected result, and more importantly, to prepare for the definitions of term algebras and initial algebras in Section 5.5.2.

Given an equational specification (S, F, E), we extend the theory Γ^{FOL} for FOL in Definition 2.51 with the following notation:

$$\forall V \,.\, t_s = t'_s \quad \equiv \quad \forall x_1 : s_1 \ldots \forall x_n : s_n \,.\, t_s =_s^{\mathsf{Formula}} t'_s$$

where $V = \{x_1 : s_1, \ldots, x_n : s_n\}$. Then, all (S, F)-terms are patterns of the corresponding sorts and all (S, F)-equations are patterns of sort $\mathsf{Formula}$. Let $\Gamma^{\mathsf{EqSpec}} = \Gamma^{\mathsf{FOL}} \cup E$ be the resulting theory for the equational specification (S, F, E).

Theorem 5.5. Under the above notation, for any equation e, the following are equivalent: (1) $\Gamma^{\mathsf{EqSpec}} \vdash e$; (2) $\Gamma^{\mathsf{EqSpec}} \vDash e$; (3) $E \vDash_{\mathsf{EQ}} e$; (4) $E \vdash_{\mathsf{EQ}} e$.

Proof. We prove that (1) $\implies$ (2) $\implies$ (3) $\implies$ (4) $\implies$ (1). Note that (1) $\implies$ (2) is the soundness of equational reasoning; (3) $\implies$ (4) is the completeness of equational deduction; and (4) $\implies$ (1) holds because matching μ-logic supports equational reasoning. Thus, we only need to prove (2) $\implies$ (3).

We show that for any (S, F, E)-algebra A, there exists a corresponding matching μ-logic model M such that for any equation e, $A \vDash_{\mathsf{EQ}} e$ iff $M \vDash e$. We define the model M as follows:

1. the carrier set $M_s = A_s$ for all $s \in S$, and $M_{\mathsf{Formula}} = \{\star\}$, where $\star$ is a distinguished dummy element;

2. $f_M(a_1, \ldots, a_n) = \{f_A(a_1, \ldots, a_n)\}$ for all $a_i \in M_i$, $1 \leq i \leq n$;

3. $\lceil a \rceil_s^{s'} = M_{s'}$ for all $a \in M_s$ and $s, s' \in S$.

Note that the above is the same model for FOL, which is known to yield the same semantics as FOL for terms [2]. Furthermore, equality has the intended semantics in matching μ-logic (Section 2.13.2). Therefore, $A \vDash_{\mathsf{EQ}} e$ iff $E \vDash e$, and we proved Theorem 5.5. QED.

5.5 DEFINING INITIAL ALGEBRA SEMANTICS

We start by defining term algebras in Section 5.5.1 and then proceed to defining initial algebras in Section 5.5.2. We discuss induction principles in Section 5.5.3 and show that induction principles can be derived as matching μ-logic theorems in Section 5.5.3.

5.5.1 Defining term algebras

Term algebras are a special case of initial algebras when there are no equations to restrict the behaviors of the operations. Theorem 2.6 gives an equivalent characterization of initiality in terms of the no-junk and no-confusion properties. We will define term algebras in matching μ-logic by defining the no-junk and no-confusion properties.

Let us first consider the no-confusion property. When there are no underlying equations, the no-confusion property takes the following simpler form:

Lemma 5.1. Let $(S, F, \emptyset)$ be an equational specification with no equations. An F-algebra A satisfies no-confusion iff (1) A_f is injective for all $f \in F$, and (2) $\mathsf{image}(A_f) \cap \mathsf{image}(A_{f'}) = \emptyset$ for all $f \neq f'$.

We can translate Lemma 5.1 into the following (No Confusion) axioms:

$$\begin{array}{ll} (\textsc{No Confusion I}) & f(x_1, \ldots, x_n) = f(x'_1, \ldots, x'_n) \to x_1 = x'_1 \wedge \cdots \wedge x_n = x'_n \\ (\textsc{No Confusion II}) & f(x_1, \ldots, x_n) \neq g(y_1, \ldots, y_m) \quad \text{if } f \neq g \end{array}$$

Next, let us consider the no-junk property. An algebra satisfies no-junk iff its carrier sets are the smallest sets closed under the operations in the signature, so we can define no-junk using the μ operator. Let us look at some examples.

Example 5.6. Consider (S, F) where $S = \{\mathsf{Nat}\}$ and $F = \{\mathsf{zero} \in F_{\epsilon,\mathsf{Nat}}, \mathsf{succ} \in F_{\mathsf{Nat},\mathsf{Nat}}\}$. We can define the carrier set of Nat as follows:

$$\top_{\mathsf{Nat}} = \mu D \,.\, \mathsf{zero} \vee \mathsf{succ}(D)$$

Intuitively, the axiom states that $\top_s$ is the smallest set D that includes zero and is closed under succ. This way, we precisely capture the set of natural numbers.

In the literature, the definition in Example 5.6 is known as *single recursion* or *direct recursion*. Generally speaking, a many-sorted signature (S, F) may include many sorts and operations, which often causes *mutual recursion*. We will convert mutual recursion to single recursion.

Example 5.7. Let $S = \{s_1, s_2\}$ and $F = \{a_1 \in F_{\epsilon,s_1}, a_2 \in F_{\epsilon,s_2}, f \in F_{s_1\,s_2,s_2}, g \in F_{s_1\,s_2,s_1}\}$. Conceptually, we would like to use the μ operator to define the mutual recursion over $\top_{s_1}$ and $\top_{s_2}$ as follows:

$$\langle \top_{s_1}, \top_{s_2} \rangle = \mu \langle D_1, D_2 \rangle \,.\, \langle a_1 \vee g(D_1, D_2), a_2 \vee f(D_1, D_2) \rangle$$

However, in matching μ-logic, μ can only bind a set variable, and not a structure such as $\langle D_1, D_2 \rangle$. To correct the definition, we replace $\langle D_1, D_2 \rangle$ by a set variable D over the pair sort $s_1 \otimes s_2$ and use the projection operation proj in Definition 5.2 to restore D_1 and D_2. The corrected definition is

$$\langle \top_{s_1}, \top_{s_2} \rangle = \mu D \,.\, \langle a_1 \vee g(D_1, D_2), a_2 \vee f(D_1, D_2) \rangle$$

where $D_1 \equiv \mathsf{proj}_1(D)$, $D_2 \equiv \mathsf{proj}_2(D)$, and $\langle \top_{s_1}, \top_{s_2} \rangle = \Phi$ is a notation for two axioms: $\top_{s_1} = \mathsf{proj}_1(\Phi)$ and $\top_{s_2} = \mathsf{proj}_2(\Phi)$. Mutual recursion over s_1 and s_2 is thus reduced to single recursion over $s_1 \otimes s_2$. In general, mutual recursion over $s_1, \dots, s_n$ can be reduced to single recursion over $s_1 \otimes \cdots \otimes s_n$.

Definition 5.4. Let (S, F) be a many-sorted signature where $F_{\epsilon,s} \neq \emptyset$ for any $s \in S$. We define

$$(\textsc{No Junk}) \qquad \langle \top_{s_1}, \dots, \top_{s_n} \rangle = \mu D \,.\, \underbrace{\langle \bigvee_{f \in F_{w,s_1}, w \in S^*} f\, D_w \,,\, \dots \,,\, \bigvee_{f \in F_{w,s_n}, w \in S^*} f\, D_w \rangle}_{\text{denoted by } F(D)}$$

where D is a set variable of sort $s_1 \otimes \cdots \otimes s_n$ and $\langle \top_{s_1}, \dots, \top_{s_n} \rangle = F(D)$ is a notation for n axioms: $\top_{s_i} = \mathsf{proj}_i(F(D))$ for $1 \leq i \leq n$.

Term algebras can then be precisely axiomatized by the (No Confusion) and (No Junk) axioms.

It is known that term algebras have a complete FOL axiomatization, such that for any FOL sentence φ, either φ or $\neg\varphi$ can be proved [41, 42]. Together with the completeness of FOL, we know that it is decidable to determine whether a given FOL sentence holds in a term algebra. The complete FOL axiomatization of term algebras is a beautiful result but understandably weaker than our result. Firstly, the FOL axiomatization does not precisely capture term algebras, but only up to elementary equivalence. In other words, there exist (nonstandard) models of the FOL axiomatization that are not term algebras, but merely satisfy the same FOL sentences as term algebras. Indeed, the FOL axiomatization allows arbitrarily large models due to the Löwenheim-Skolem theorem [43], while term algebras must be countable. In addition, the FOL axiomatization is not extensible. For example, one cannot take the complete axiomatization of zero and succ and extend it with plus, mult, and their Peano (equational) axioms, and hope to get a complete FOL axiomatization of natural numbers with addition and multiplication—the completeness is lost in the extension. In contrast, our matching μ-logic axiomatization of term algebras using (No Junk) and (No

CONFUSION) precisely captures term algebras. We can extend it with equations to further define initial algebras, which will be discussed in Section 5.5.2.

5.5.2 Defining initial algebras

Let (S, F, E) be an equational specification. Recall that $\simeq_E$ is the smallest relation that includes the identity relation and all the equations in E, and is closed under converse, composition, and congruence w.r.t. all the operations in F (Proposition 2.4). Thus, we can define $\simeq_E$ using the μ operator.

Specifically, for every $s \in S$, we introduce a constant symbol $\mathsf{Eq}_s \in \Sigma_{\epsilon, s\otimes s}$ or simply Eq, and add the following axioms:

$$\begin{aligned}
\varphi_1 \precsim \varphi_2 &\equiv \forall x_1 \,.\, x_1 \in \varphi_1 \to \exists x_2 \,.\, x_2 \in \varphi_2 \,.\, \langle x_1, x_2\rangle \in \mathsf{Eq} \\
\varphi_1 \simeq \varphi_2 &\equiv \varphi_1 \precsim \varphi_2 \wedge \varphi_2 \precsim \varphi_1 \\
R^{-1} &\equiv \mathsf{converseRel}\ R \\
R_1 \circ R_2 &\equiv \mathsf{composeRel}\ R_1\ R_2
\end{aligned}$$

$$\begin{aligned}
&(\text{IDENTITY}) && \mathsf{idRel} = \bigvee_{s \in S} \exists x : s \,.\, \langle x, x\rangle \\
&(\text{CONVERSE}) && R^{-1} = \exists x \,.\, \exists y \,.\, \langle y, x\rangle \wedge (\langle x, y\rangle \in R) \\
&(\text{COMPOSITION}) && R_1 \circ R_2 = \exists x \,.\, \exists y \,.\, \exists z \,.\, \langle x, z\rangle \wedge (\langle x, y\rangle \in R_1 \wedge \langle y, z\rangle \in R_2) \\
&(\text{CONGRUENCE}) && \mathsf{congRel}\ R = \bigvee_{f \in F_{s_1 \ldots s_n, s}} \exists x_1, y_1 : s_1 \ldots \exists x_n, y_n : s_n \,. \\
& && \qquad \langle f(x_1, \ldots, x_n), f(y_1, \ldots, y_n)\rangle \wedge \bigwedge_{1 \le i \le n} \langle x_i, y_i\rangle \in R \\
&(\text{EQUIVALENCE}) && \mathsf{Eq} = \mu R \,.\, \mathit{idRel} \vee R^{-1} \vee (R \circ R) \vee (\mathsf{congRel}\ R) \vee \bigvee_{(\forall V \,.\, t = t') \in E} \exists V \,.\, \langle t, t'\rangle
\end{aligned}$$

Here, idRel is the identity relation; R^{-1} is the converse relation of R; $R_1 \circ R_2$ is composition of R_1 and R_2; and $\mathsf{congRel}\ R$ is the relation obtained by propagating R through all operations in F. The axiom (EQUIVALENCE) states that Eq is the smallest relation that includes idRel and all equations in E, and is closed under converse, composition, and congruence. We write $\varphi_1 \precsim \varphi_2$ to mean that φ_1 is contained in φ_2 modulo Eq, and $\varphi_1 \simeq \varphi_2$ to mean that φ_1 and φ_2 are equal modulo Eq. This way, initial E-algebras are precisely captured by the above axioms. Note that we distinguish two different equalities: one is the pure syntactic equality (written $t = t'$) and the other is $\simeq_E$-equivalence (written $t \simeq t'$). The latter corresponds to the equality in the quotient term algebra $T_{F/E}$.

5.5.3 Deriving induction principles as matching μ-logic theorems

Initial algebra reasoning is a synonym for induction. Various inductive techniques in formal program verification flourished in the 1960s [46, 47, 48, 49, 50, 51, 52]. Later, it was discovered that initiality, or more precisely, the no-junk property is what powers induction and the induction-based proof techniques in initial algebras [20, Proposition 16]. If an algebra A satisfies no-junk, all elements of A can be represented by some terms, and thus the (unique) morphism $f_A \colon T_{F/E} \to A$ is surjective. Inductive principles on $T_{F/E}$ are then mapped to A through f_A, whose surjectivity guarantees that all elements in A are covered in the inductive reasoning. Various induction principles have been adopted as proof-theoretical alternatives to initiality (see, e.g., [20, Section 4.4]) and have led to practical tools [32].

In Section 5.5.1, we define the no-junk property using (No Junk). Matching μ-logic also has one proof rule—(Knaster Tarski)—which is dedicated to fixpoint reasoning. In what follows, we show that inductive reasoning can be obtained by combining (No Junk) and (Knaster Tarski). That is to say, induction is a special case of matching μ-logic reasoning in the theory of initial algebras:

$$\boxed{\text{Induction}} = \boxed{(\textsc{No-Junk})} + \boxed{(\textsc{Knaster Tarski})}$$

Let us consider natural numbers built from zero and succ. We have two Peano axioms: $E^{\mathsf{Nat}} = \{\forall x : \mathsf{Nat} \,.\, \mathsf{plus}(x, \mathsf{zero}) \simeq x$ and $\forall x, y : \mathsf{Nat} \,.\, \mathsf{plus}(x, \mathsf{succ}(y)) \simeq \mathsf{succ}(\mathsf{plus}(x, y))\}$. Let us prove the following property:

$$\forall y : \mathsf{Nat} \,.\, \mathsf{plus}(\mathsf{zero}, y) \simeq y \tag{5.2}$$

Note that (5.2) does not hold in an arbitrary algebra that satisfies the Peano axioms. Consider, for example, an algebra with only two elements $\{0, \star\}$, where zero is interpreted as 0, succ is interpreted as the identity function on $\{0, \star\}$, and plus is interpreted as the binary function that returns its first argument. In this algebra, both axioms hold, but not (5.2).

Property (5.2) holds if we consider the initial algebra of E^{Nat}. However, by default, initial algebra semantics does not distinguish constructors and defined functions. Instead, it treats them equally as operations. It results in unnecessarily tedious inductive proofs, because induction cases are created for all operators, even for defined functions (see, e.g., [53, Section 2.4]). For example, when we prove (5.2) and apply inductive reasoning on y, we have three cases rather than two, where the extra one is for plus. This is not expected. To carry out the usual inductive reasoning with only cases for zero and succ, we need to prove that plus is a defined function, i.e., it does not effectively create new (ground) terms (Step 1).

After that, we apply structural induction on (5.2) w.r.t. zero and succ only (Step 2), and prove all the sub-goals (Step 3).

In many initial algebra semantics papers [54, 55, 56, 57, 58, 59] and tools [32], Step 1 is proved by noting that the two Peano equations, when oriented from left to right, become rewrite rules that reduce the size of the sub-terms whose top-level operation is plus. Any ground term that contains plus can be rewritten to a canonical term without plus. Therefore, plus is a defined function. This technique, called *sufficient completeness*, goes back to [60] and is further developed and implemented in the above-mentioned works.

In practice, inductive equational theorem provers allow users to explicitly declare constructors and defined functions, following one (or both) of the following aesthetically different but ultimately equivalent approaches:

1. To declare a sub-signature of constructors (supported by Maude [32] and proof assistants such as Coq [116]).

2. To define a sub-specification of constructors and import it in a "protected" mode (supported by OBJ [61], CafeOBJ [62], and Maude [32]).

Either way, initiality is defined only for constructors. Defined functions must be proved well-defined. Both approaches are extensions to the vanilla equational specifications in Section 2.5: (1) adds constructor signatures and (2) adds a module system. What is not an extension is the following axiomatic matching μ-logic approach, where the statement "plus is a defined function" is expressed and derived within matching μ-logic: [1]

Theorem 5.8. $\vdash \underbrace{(\mu D \,.\, \mathsf{zero} \vee \mathsf{succ}(D) \vee \mathsf{plus}(D, D))}_{\text{equals to } \top_{\mathsf{Nat}} \text{ by axiom (No Junk)}} \simeq (\mu D \,.\, \mathsf{zero} \vee \mathsf{succ}(D))$

That is, the smallest set generated by $\{\mathsf{zero}, \mathsf{succ}, \mathsf{plus}\}$ equals to the one generated by $\{\mathsf{zero}, \mathsf{succ}\}$, modulo E^{Nat}. Theorem 5.8, which accomplishes Step 1, is a theorem that is formally derivable using the existing proof system, requiring no reasoning outside the logic, which is in sharp contrast to the classical initial algebra semantics approaches and tools.

An advantage of specifying defined functions by theorems (such as Theorem 5.8) is that we can reason about abstract datatypes (ADTs) using different but equivalent constructor sets. For example, lists can be defined using nil and cons, or using nil, one-element lists, and concatenation. Such flexibility is not possible in frameworks that enforce explicit specification of the constructors for each ADT.

[1] A similar result holds for the general case, when E includes equations for constructors and $E' \supseteq E$ further includes those for defined functions. The well-definedness of the defined functions can be proved by showing that E-equality is preserved, i.e., $\simeq_E$ equals to $\simeq_{E'}$, where $\simeq_E$ and $\simeq_{E'}$ are least fixpoint patterns as the right- and left-hand sides of Theorem 5.8.

After Step 1, we carry out Step 2, which is to apply structural induction on y in (5.2). It generates two sub-goals:

$$\mathsf{plus}(\mathsf{zero},\mathsf{zero}) \simeq \mathsf{zero} \tag{5.3}$$

$$\forall z : \mathsf{Nat} \,.\, \mathsf{plus}(\mathsf{zero}, z) \simeq z \to \mathsf{plus}(\mathsf{zero}, \mathsf{succ}(z)) \simeq \mathsf{succ}(z) \tag{5.4}$$

where (5.3) is the base case and (5.4) is the induction case. In matching μ-logic, the above inductive proof is carried out, within the logic, using (Knaster Tarski). Specifically, let

$$\Psi \equiv \exists y : \mathsf{Nat} \,.\, y \wedge (\mathsf{plus}(\mathsf{zero}, y) \simeq y) \tag{5.5}$$

be the pattern that is matched by all y that satisfy (5.2). Then,

$$\begin{aligned}
&\vdash \forall y : \mathsf{Nat} \,.\, \mathsf{plus}(\mathsf{zero}, y) \simeq y && \text{iff} \\
&\vdash \top_{\mathsf{Nat}} \to \Psi && \text{iff} \\
&\vdash (\mu D \,.\, \mathsf{zero} \vee \mathsf{succ}(D)) \to \Psi && \text{if} \quad \text{// by (Knaster Tarski)} \\
&\vdash \mathsf{zero} \to \Psi \text{ and } \vdash \mathsf{succ}(\Psi) \to \Psi && \text{iff} \\
&\vdash \mathsf{plus}(\mathsf{zero}, \mathsf{zero}) \simeq \mathsf{zero} \quad \text{and} && \\
&\vdash \forall y : \mathsf{Nat} \,.\, \mathsf{plus}(\mathsf{zero}, y) \simeq y \to \mathsf{plus}(\mathsf{zero}, \mathsf{succ}(y)) \simeq \mathsf{succ}(y) &&
\end{aligned} \tag{5.6}$$

Finally, we carry out Step 3 and prove (5.3) and (5.4) by equational reasoning, which is a special instance of matching logic reasoning. Therefore, we formally derived (5.2) as a matching μ-logic theorem using the proof system.

Theorem 5.9. Under the above notation, for any pattern Ψ of sort Nat,

$$\frac{\mathsf{zero} \to \Psi \quad \mathsf{succ}(\Psi) \to \Psi}{\top_{\mathsf{Nat}} \to \Psi}$$

Proof. Use (Knaster Tarski) and the definition of $\top_{\mathsf{Nat}}$. QED.

Theorem 5.9 is the logical incarnation of Peano induction in matching μ-logic, where Ψ is any property that we want to (inductively) prove for natural numbers. The first premise, $\vdash \mathsf{zero} \to \Psi$, states that zero satisfies property Ψ. The second premise states that the following induction case holds:

Lemma 5.2. Under the above notation, $\vdash \mathsf{succ}(\Psi) \to \Psi$ iff $\vdash \forall x : \mathsf{Nat} \,.\, (x \in \Psi \to \mathsf{succ}(x) \in \Psi)$. Note that the right-hand side is exactly the induction case of Peano induction.

Intuitively, the right-hand side states that Ψ is closed under succ; that is, if we start with any x that satisfies Ψ and apply succ to it, $\mathsf{succ}(x)$ still satisfies Ψ. Hence, if we apply succ to Ψ, which, by definition, is matched by all the numbers that satisfy Ψ, the result $\mathsf{succ}(\Psi)$ is still included by Ψ. And that is exactly the left-hand side.

It is not a coincidence that the proof rule (Knaster Tarski) has such a close connection to induction principles. In our view, induction principles are instances of the Knaster-Tarski fixpoint theorem (Theorem 2.1, of which (Knaster Tarski) is a logical encoding. It is particularly interesting to see that such a connection can be so elegantly expressed within matching μ-logic as formal proofs, by the following theorem.

Theorem 5.10. Under the notation in Section 5.5.1, for any pattern Ψ of sort $s_1 \otimes \cdots \otimes s_n$,

$$\frac{F\,\Psi \to \Psi}{\langle \top_{s_1}, \ldots, \top_{s_n} \rangle \to \Psi} \tag{5.7}$$

which is an abbreviation for the following:

$$\frac{\bigvee_{f \in F_{w,s_1}, w \in S^*} f\,\Psi_w \to \Psi_{s_1} \quad \cdots \quad \bigvee_{f \in F_{w,s_n}, w \in S^*} f\,\Psi_w \to \Psi_{s_n}}{\top_{s_1} \to \Psi_{s_1} \text{ and } \top_{s_2} \to \Psi_{s_2} \text{ and } \ldots \text{ and } \top_{s_n} \to \Psi_{s_n}}$$

where $\Psi_{s_i} \equiv \mathsf{proj}_i(\Psi)$ is the i-th projection of Ψ and $f\,\Psi_w \equiv (f\,\Psi_{s'_1} \cdots \Psi_{s'_m})$ for $w = s'_1 \ldots s'_m$.

Proof. Use (Knaster Tarski) and (No Junk). QED.

To conclude, initial algebras can be precisely captured by matching μ-logic by defining the no-junk and no-confusion properties. Inductive reasoning is a special case of matching μ-logic reasoning in the theory of initial algebras, and induction principles can be derived as matching μ-logic theorems using the proof system.

5.6 DEFINING SECOND-ORDER LOGIC

To define second-order logic (SOL), we need to define powersets. More specifically, for sorts $s_1, \ldots, s_n$ we define a new sort $2^{s_1 \otimes \cdots \otimes s_n}$, called the *power sort of* $s_1, \ldots, s_n$, with the intuition that $M_{2^{s_1 \otimes \cdots \otimes s_n}} = \mathcal{P}(M_{s_1} \times \cdots \times M_{s_n})$, which is the set of relations over $M_{s_1}, \ldots, M_{s_n}$. That is to say, every element of sort $2^{s_1 \otimes \cdots \otimes s_n}$ is a relation over $s_1 \times \cdots \times s_n$. Then, we can reduce second-order quantification over $s_1 \times \cdots \times s_n$ to first-order quantification over $2^{s_1 \otimes \cdots \otimes s_n}$.

We first show the definition of powersets in Section 5.6.1. Then we show the definition of monadic SOL in Section 5.6.2, where we reduce (monadic) second-order quantification over one sort s to first-order quantification over 2^s. Finally, we consider full SOL in Section 5.6.

5.6.1 Defining powersets

Matching μ-logic has set variables that range over the subsets of the underlying carrier set(s). Recall that $M \vDash \varphi$ iff $|\varphi|_{M,\rho} = M$ for all ρ. This means that if an axiom φ has free set variables, then they are, semantically speaking, universally quantified. This way, we can write matching μ-logic axioms that have the same expressive power as monadic SOL, where all predicate variables are universally quantified at the top. We can use this feature to define powersets.

Definition 5.5. Let s be a sort. We define a new sort 2^s called the *power sort of* s and a symbol $\mathsf{extension} \in \Sigma_{2^s,s}$ called the *extension symbol.* We use $\alpha, \beta, \ldots$ to denote element variables of 2^s. We define the following axioms:

$$\begin{aligned} &(\textsc{Powerset}) && \exists \alpha : 2^s \,.\, \mathsf{extension}(\alpha) = X : s \\ &(\textsc{Extensionality}) && \forall \alpha : 2^s \,.\, \forall \beta : 2^s \,.\, \mathsf{extension}(\alpha) = \mathsf{extension}(\beta) \to \alpha = \beta \end{aligned}$$

Note that $X : s$ is a free set variable in (Powerset).

To understand Definition 5.5, let us consider an arbitrary model M where M_s and M_{2^s} are the carrier sets of s and 2^s, respectively, and $\mathsf{extension}_M : M_{2^s} \to \mathcal{P}(M_s)$ is the interpretation of $\mathsf{extension}$ in M. The axiom (Powerset) states that for any $X \subseteq M_s$ there exists there exists $\alpha \in M_{2^s}$ such that $\mathsf{extension}_M(\alpha) = X$. In other words, $\mathsf{extension}_M$ is surjective. On the other hand, (Extensionality) states that $\mathsf{extension}_M$ is injective. Therefore, $\mathsf{extension}_M$ is a bijection from M_{2^s} to $\mathcal{P}(M_s)$. Its reverse, called *intension*, is given by

$$\mathsf{intension}(\varphi_s) \equiv \exists \alpha : 2^s \,.\, \alpha \wedge (\mathsf{extension}(\alpha) = \varphi_s)$$

Note that $\mathsf{intension}(\varphi)$ is a singleton pattern, i.e., it is matched by exactly one element. This is guaranteed by (Extensionality).

Note the difference between α and $\mathsf{extension}(\alpha)$. The former is an element variable of sort 2^s and is matched by one element in M_{2^s}, which, according to the bijection above, represents a set of elements in M_s. The latter is a pattern of sort s, which also represents a set of elements in M_s that match it. The difference is that α is regarded as an element while

$\mathsf{extension}(\alpha)$ is regarded as a set. The term "extension" has a similar meaning in logic and philosophy; an extension of a concept consists of the things to which it applies. Here, we see α as a concept and $\mathsf{extension}(\alpha)$ as its extension.

The above definition of powersetsis not possible in FOL, because by the Löwenheim-Skolem theorem [43], if a FOL theory has infinite models, then it has a countable model. However, using powersets, we can enforce uncountable models by first enforcing an infinite model and considering its powerset. For example, we can define a sort Nat with two functions zero and succ, and define their injectivity axioms $\mathsf{zero} \neq \mathsf{succ}(x)$ and $\mathsf{succ}(x) = \mathsf{succ}(y) \to x = y$. These axioms enforce infinite models because the following infinitely-many terms zero, $\mathsf{succ}(\mathsf{zero})$, $\mathsf{succ}(\mathsf{succ}(\mathsf{zero}))$, etc., must be different. If powersets could been completely axiomatized in FOL, then we could define the power sort 2^{Nat}, whose carrier set must be uncountable, contradicting the Löwenheim-Skolem theorem. The reason why powersets can be precisely defined in matching μ-logic is because matching μ-logic has set variables, and by writing axioms with free set variables, we obtain the expressive power of (monadic) universal second-order quantification.

5.6.2 Defining monadic SOL

Monadic SOL, abbreviated as MSO, is the instance of SOL with only unary/monadic predicate variables. Let us fix a MSO signature (S, C, Π) where S is a set of sorts, C is an S-indexed set of constant symbols, and Π is an S^*-indexed set of predicate symbols. Let $EV = \{EV_s\}_{s \in S}$ be an S-indexed set of element variables and $PV = \{PV_s\}_{s \in S}$ be an S-indexed set of unary predicate variables.

We define the corresponding matching μ-logic signature $(S^{\mathsf{MSO}}, \Sigma^{\mathsf{MSO}})$ and theory Γ^{MSO} for MSOL. Let

$$\begin{aligned} S^{\mathsf{MSO}} &= S \cup \{2^s \mid s \in S\} \cup \{\mathsf{Formula}\} \\ \Sigma^{\mathsf{MSO}} &= \Sigma^{\mathsf{powersort}} \cup C \cup \Pi \\ \Gamma^{\mathsf{MSO}} &= \Gamma^{\mathsf{powersort}} \cup \Gamma^{\mathsf{function}(C)} \cup \Gamma^{\mathsf{predicate}(\Pi)} \end{aligned}$$

That is, S^{MSO} includes all the sorts in S and their corresponding power sorts, plus a distinguished sort $\mathsf{Formula}$ for MSO formulas. The symbol set Σ^{MSO} includes the necessary symbols for powersorts and all the constant and predicate symbols of MSO. The theory Γ^{MSO} includes the necessary axioms for powersorts, the function axioms for the constant symbols in C, and the predicate axioms for the predicate symbols in Π.

Next, we show that MSO formulas are patterns of sort $\mathsf{Formula}$. We include all unary predicate variables in PV as matching μ-logic element variables of the corresponding power sorts. More specifically, for every $R \in PV_s$, we add $R : 2^s$ or simply R as a matching μ-logic element variable of sort 2^s. Then, we define the following notations:

$$R(t_s) \equiv t_s \in_s^{\mathsf{Formula}} \mathsf{extension}(R) \qquad \text{with } R \in PV_s \text{ and } t_s \text{ is a term}$$
$$\exists R \,.\, \varphi \equiv \exists R : 2^s \,.\, \varphi \qquad \text{with } R \in PV_s$$

Under the above notation, all MSO formulas are patterns of sort $\mathsf{Formula}$.

Theorem 5.11. For any MSO formula φ, $\vDash_{\mathsf{SOL}} \varphi$ iff $\Gamma^{\mathsf{MSO}} \vDash \varphi$.

Proof. Note that there is a one-to-one correspondence between the SOL models and the matching μ-logic Γ^{MSO}-models. For any SOL model $M = (\{M_s\}_{s\in S}, \{c_M\}_{c\in C}, \{\pi_M\}_{\pi\in\Pi})$, we define the corresponding matching μ-logic model $M' = (\{M'_s\}_{s\in S^{\mathsf{MSO}}}, \{\sigma_{M'}\}_{\sigma\in\Sigma^{\mathsf{MSO}}})$ where

1. $M'_s = M_s$ for $s \in S$;

2. $M'_{2^s} = \mathcal{P}(M_s)$;

3. $M'_{\mathsf{Formula}} = \{\star\}$;

4. $c_{M'} = \{c_M\}$ for $c \in C$;

5. $\pi_{M'}(a_{s_1}, \ldots, a_{s_n}) = \{\star\}$ iff $\pi_M(a_{s_1}, \ldots, a_{s_n})$ holds, for $\pi \in \Pi_{s_1 \ldots s_n}$ and $a_{s_i} \in M_{s_i}$ for $1 \le i \le n$.

6. $\mathsf{extension}_{M'}(A_s) = A_s$ for $A_s \subseteq M_s$.

Note that (2) and (6) are enforced by the axioms for power sorts. By structural induction, we can show that $M, \rho \vDash_{\mathsf{SOL}} \varphi$ iff $|\varphi|_{M',\rho} = \{\star\}$, for any MSO formula φ. Note that the FOL cases have been considered in Section 2.13.2, so here we only need to consider two cases: φ has the form $R(t_s)$ or $\exists R \,.\, \varphi_1$. If φ has the form $R(t_s)$, $M, \rho \vDash_{\mathsf{SOL}} R(t_s)$ iff $\rho(R)(\bar{\rho}(t_s))$ holds, iff $|t_s|_{M',\rho} \in \rho(R)$, iff $|t_s \in \mathsf{extension}(R)|_{M',\rho} = \{\star\}$ by the semantics of $\in$, iff $|R(t_s)|_{M',\rho} = \{\star\}$. If φ has the form $\exists R \,.\, \varphi$, $M, \rho \vDash_{\mathsf{SOL}} \exists R \,.\, \varphi_1$ iff there exists $\alpha_R \subseteq M_s$ such that $M, \rho[\alpha_R/R] \vDash_{\mathsf{SOL}} \varphi_1$, iff there exists $\alpha_R \in M'_{2^s}$ such that $|\varphi_1|_{M',\rho[\alpha_R/R]} = \{\star\}$ by the induction hypothesis, iff $|\exists R \,.\, \varphi_1|_{M',\rho} = \{\star\}$. Since M and ρ are arbitrarily chosen, $\vDash_{\mathsf{SOL}} \varphi$ iff $\Gamma^{\mathsf{MSO}} \vDash \varphi$ for any MSO formula φ. QED.

5.6.3 Defining full SOL

We extend Theorem 5.11 to full SOL, by allowing predicate variables of any arities. The idea is similar. For any predicate variable R of sort $s_1 \times \cdots \times s_n$, we add it as an element variable of sort $2^{s_1 \otimes \cdots \otimes s_n}$. Then, second-order quantification over R of sort $s_1 \times \cdots \times s_n$ becomes first-order quantification over (element variable) R of the power sort $2^{s_1 \otimes \cdots \otimes s_n}$. This is because for any matching μ-logic model M, $M_{2^{s_1 \otimes \cdots \otimes s_n}}$ is isomorphic to $\mathcal{P}(M_{s_1} \times \cdots \times M_{s_n})$, which is exactly the set of relations over $M_{s_1}, \ldots, M_{s_n}$.

Let us fix a SOL signature (S, C, Π). Let $EV = \{EV_s\}_{s \in S}$ be an S-indexed set of element variables and $PV = \{PV_s\}_{s \in S}$ be an S^+-indexed set of predicate variables. The corresponding matching μ-logic signature $(S^{\mathsf{SOL}}, \Sigma^{\mathsf{SOL}})$ and theory Γ^{SOL} for SOL are defined as follows:

$$
\begin{aligned}
S^{\mathsf{SOL}} &= S \cup \{2^{s_1 \otimes \cdots \otimes s_n} \mid s_1, \ldots, s_n \in S, n \geq 1\} \cup \{\mathsf{Formula}\} \\
\Sigma^{\mathsf{SOL}} &= \Sigma^{\mathsf{powersort}} \cup C \cup \Pi \\
\Gamma^{\mathsf{SOL}} &= \Gamma^{\mathsf{powersort}} \cup \Gamma^{\mathsf{function}(C)} \cup \Gamma^{\mathsf{predicate}(\Pi)}
\end{aligned}
$$

Furthermore, for every $R \in PV_{s_1 \ldots s_n}$, we add it as a matching μ-logic element variable $R : 2^{s_1 \otimes \cdots \otimes s_n}$ or simply R, and define the following notation:

$$
\begin{aligned}
R(t_{s_1}, \ldots, t_{s_n}) &\equiv \langle t_{s_1}, \ldots, t_{s_n} \rangle \in^{\mathsf{Formula}}_{s_1 \otimes \cdots \otimes s_n} \mathsf{extension}(R) \\
\exists R \,.\, \varphi &\equiv \exists R : 2^{s_1 \otimes \cdots \otimes s_n} \,.\, \varphi
\end{aligned}
$$

Then all SOL formulas are patterns of sort $\mathsf{Formula}$.

Theorem 5.12. For any SOL formula φ, $\vDash_{\mathsf{SOL}} \varphi$ iff $\Gamma^{\mathsf{SOL}} \vDash \varphi$.

Proof. The proof is similar to Theorem 5.11. We show that $M, \rho \vDash_{\mathsf{SOL}} \varphi$ iff $|\varphi|_{M',\rho} = \{\star\}$, where M' is the corresponding matching μ-logic model of a given SOL model M. The proof is also by structural induction on φ, and we only need need to consider one more case, which is when φ has the form $R(t_{s_1}, \ldots, t_{s_n})$ for $R \in PV_{s_1 \ldots s_n}$. In this case, $M, \rho \vDash_{\mathsf{SOL}} R(t_{s_1}, \ldots, t_{s_n})$ iff $\rho(R)(\bar{\rho}(t_{s_1}), \ldots, \bar{\rho}(t_{s_n}))$ holds, iff $|\langle t_{s_1}, \ldots, t_{s_n} \rangle|_{M',\rho} \in |R|_{M',\rho}$, iff $|R(t_{s_1}, \ldots, t_{s_n})|_{M',\rho} = \{\star\}$. All the other cases are the same as Theorem 5.11. QED.

5.7 DEFINING TRANSITION SYSTEMS

We show how to define transition systems in matching μ-logic. Let L be a label set. An L-labeled transition system is a tuple $T = (S, \{\xrightarrow{a}\}_{a \in L})$, where $\xrightarrow{a} \subseteq S \times S$ is a binary transition relation for every $a \in L$. Let us define the corresponding matching μ-logic signature $(S^{\mathsf{TS}}, \Sigma^{\mathsf{TS}})$ for L-labeled transition systems as follows:

$$\begin{aligned} S^{\mathsf{TS}} &= \{\mathsf{State}\} \\ \Sigma^{\mathsf{TS}} &= \{\bullet_a \in \Sigma^{\mathsf{TS}}_{\mathsf{State},\mathsf{State}} \mid a \in L\} \end{aligned}$$

Here, $\bullet_a$ is a unary symbol, called *(one-path) next*, which captures the transition relation $\xrightarrow{a}$, with the intuition that $s \in \bullet_a(s')$ iff $s \xrightarrow{a} s'$ for $s, s' \in S$. When a is not important or we only consider unlabeled transition systems, we drop the subscript and simply write $\bullet_a \in \Sigma^{\mathsf{TS}}\mathsf{State}, \mathsf{State}$. In other words, there is a one-to-one correspondence between L-labeled transition systems and $(S^{\mathsf{TS}}, \Sigma^{\mathsf{TS}})$-models, given as follows. For every $T = (S, \{\xrightarrow{a}\}_{a \in L})$, we can define a corresponding $(S^{\mathsf{TS}}, \Sigma^{\mathsf{TS}})$-model M with $M_{\mathsf{State}} = S$ and $M_{(\xrightarrow{a})}(s') = \{s \in S \mid s \xrightarrow{a} s'\}$ for $s' \in S$.

It may be a little counterintuitive that the "one-path next" symbol $\bullet_a$ returns all the predecessors of a given state. This is because the "next" semantics happens on patterns, not on states. Let us look at the following state transitions:

$$\begin{array}{ccccccccc} \cdots & s & \xrightarrow{a} & s' & \xrightarrow{a} & s'' & \cdots & // \text{ states} \\ & \bullet_a \bullet_a \varphi & & \bullet_a \varphi & & \varphi & & // \text{ patterns} \end{array}$$

Suppose s'' satisfies, or matches φ. Then it is natural that s', which is a predecessor of s'', matches $\bullet_a \varphi$, because φ holds in one of the next states of s'. Similarly, s matches $\bullet_a \bullet_a \varphi$ because φ holds in one of the next, next states. Because we want $\bullet_a \varphi$ to mean that "φ holds next", the interpretation function $M_{(\xrightarrow{a})}$ must take us backward in terms of state transitions.

The dual of $\bullet_a \varphi$ is $\circ_a \varphi$, called *all-path next*, defined by $\circ_a \varphi \equiv \neg \bullet_a \neg \varphi$. Other derived operators are as follows:

$$\begin{aligned} \text{"eventually" } \diamond_a \varphi &\equiv \mu X \,.\, \varphi \vee \bullet_a X \\ \text{"always" } \Box_a \varphi &\equiv \nu X \,.\, \varphi \wedge \circ_a X \\ \text{"until" } \varphi_1 \, U_a \, \varphi_2 &\equiv \mu X \,.\, \varphi_2 \vee (\varphi_1 \wedge \bullet_a X) \\ \text{"well-founded" } \mathsf{WF}_a &\equiv \mu X \,.\, \circ_a X \quad // \text{ no infinite paths} \end{aligned}$$

Again, we feel free to drop the subscript a when it is not important or we only consider unlabeled transition systems.

Proposition 5.13. *Let S be a set of states and $\rightarrow \subseteq S \times S$ be a transition relation. Let M be the corresponding $(S^{\mathsf{TS}}, \Sigma^{\mathsf{TS}})$-model, then*

- *$s \in |\bullet\varphi|_{M,\rho}$ iff there exists $t \in S$ such that $s \to t$ and $t \in |\varphi|_{M,\rho}$; in particular, $s \in |\bullet\top|_{M,\rho}$ iff s is not a deadlock, i.e., s has a successor;*

- *$s \in |\circ\varphi|_{M,\rho}$ iff for all $t \in S$ such that $s \to t$, $t \in |\varphi|_{M,\rho}$; in particular, $s \in |\circ\bot|_{M,\rho}$ iff s is a deadlock;*

- *$s \in |\diamond\varphi|_{M,\rho}$ iff there exists $t \in S$ such that $s \to^* t$, $t \in |\varphi|_{M,\rho}$;*

- *$s \in |\Box\varphi|_{M,\rho}$ iff for all $t \in S$ such that $s \to^* t$, $t \in |\varphi|_{M,\rho}$;*

- *$s \in |\varphi_1 \, U \, \varphi_2|_{M,\rho}$ iff there exists $n \geq 0$ and $t_1, \dots, t_n \in S$ such that $s \to t_1 \to \cdots \to t_n$, $t_n \in |\varphi_2|_{M,\rho}$, and $s, t_1, \dots, t_{n-1} \in |\varphi_1|_{M,\rho}$;*

- *$s \in |\mathsf{WF}|_{M,\rho}$ iff s is well-founded, i.e., there is no infinite sequence $t_1, t_2, \dots \in S$ with $s \to t_1 \to t_2 \to \dots$;*

where $(\to^*) = \bigcup_{i \geq 0}(\to^i)$ is the reflexive transitive closure of $\to$.

5.8 DEFINING MODAL μ-CALCULUS

Modal μ-calculus is an instance of matching μ-logic when we fix the signature to be $(S^{\mathsf{TS}}, \Sigma^{\mathsf{TS}})$ in Section 5.7 and let $\Gamma^\mu = \emptyset$ be the empty theory. We add all atomic propositions of modal μ-calculus as set variables, then all modal μ-calculus formulas are patterns of sort State.

Theorem 5.14. The following properties are equivalent for all modal μ-calculus formulas φ: (1) $\models_\mu \varphi$; (2) $\vdash_\mu \varphi$; (3) $\Gamma^\mu \vdash \varphi$; (4) $\Gamma^\mu \models \varphi$; (5) $M \models \varphi$ for all Σ^{TS}-models M such that $M \models \Gamma^\mu$; (6) $T \models_\mu \varphi$ for all transition systems T.

Proof. The proof is straightforward. (1) $\Longrightarrow$ (2) is by Theorem 2.7; (2) $\Longrightarrow$ (3) is because all modal μ-calculus proof rules in Figure 2.3 are derivable in matching μ-logic (Proposition 3.6); (3) $\Longrightarrow$ (4) is by Theorem 4.1; follows by the soundness of matching μ-logic. (4) $\Longrightarrow$ (5) is by definition; (5) $\Longrightarrow$ (6) is by proving that for any transition system T, $T, a \models_{L\mu} \varphi$ iff $a \in |\varphi|_{M,\rho}$, where M is the corresponding Σ^{TS}-model; this is proved by applying structural induction on φ; Finally, (6) $\Longrightarrow$ (1) follows by definition. QED.

Therefore, modal μ-calculus with multiple modalities can be regarded as an instance of matching μ-logic where the signature is Σ^{TS} and the theory Γ^μ is empty. It is worth mentioning that modal μ-calculus considers only unary modal modalities and they are only required to obey the usual (K) and (N) rules, while matching μ-logic allows polyadic and even many-sorted symbols while still obeying the desired (K) and (N) rules (see Proposition 3.6), allows arbitrary further constraining axioms in theories, and also allows quantification over element variables and many-sorted universes. It thus suggests that matching μ-logic may offer a unifying playground to specify and reason about transition systems, by means of Σ^{TS}-theories/models. We can define various temporal modalities and dynamic operations as notation using the basic "one-path next" symbol $\bullet \in \Sigma^{\mathsf{TS}}$ and the μ operator, without a need to extend the logic. We can restrict the underlying transition systems using axioms, without a need to modify or extend the proof system. In Sections 5.9 to 5.11, we show that by adding proper axioms and introducing good notation, we can define various logics for specifying and reasoning about dynamic behaviors of programs and computing systems, such as linear temporal logic (LTL), computation tree logic (CTL), dynamic logic (DL), and reachability logic (RL).

5.9 DEFINING TEMPORAL LOGICS

Since matching μ-logic can define modal μ-calculus, it is not surprising that it can also define various temporal logics such as LTL and CTL as theories whose axioms constrain the underlying transition relations. What is interesting, in our view, is that the resulting theories are simple, intuitive, and faithfully capture both the semantics and formal proofs of these temporal logics.

5.9.1 Defining infinite-trace LTL

We have seen the syntax and semantics of infinite-trace LTL in Section 2.9.1. Note that the infinite-trace LTL syntax, namely the following

$$\varphi ::= p \in AP \mid \varphi \land \varphi \mid \neg\varphi \mid \circ\varphi \mid \varphi\, U\, \varphi$$

has already been subsumed by matching μ-logic. As for models, infinite-trace LTL requires infinite traces, so the underlying transition relations are linear (i.e., $s \xrightarrow{T} s'$ and $s \xrightarrow{T} s''$ implies $s' = s''$) and infinite (i.e. deadlock-free: for every s there is s' such that $s \xrightarrow{T} s'$). To capture these two characteristics, we add two axioms:

$$(\textsc{Inf})\ \ \bullet\top \qquad\qquad\qquad (\textsc{Lin})\ \ \bullet X \to \circ X$$

and denote the resulting Σ^{TS}-theory as Γ^{infLTL}. Note that by (SUBSTITUTION) in Figure 4.1 we can prove from axiom (LIN) that $\bullet\varphi \to \circ\varphi$ for all patterns φ. Intuitively, (INF) forces all states s to have at least one successor, and thus all traces can be extended to an infinite trace, and (LIN) forces all states s to have only a linear future.

Theorem 5.15 shows that Γ^{infLTL} captures the semantics and formal proofs of infinite-trace LTL.

Theorem 5.15. The following properties are equivalent for all infinite-trace LTL formulas φ: (1) $\vdash_{\mathsf{infLTL}} \varphi$; (2) $\vDash_{\mathsf{infLTL}} \varphi$; (3) $\Gamma^{\mathsf{infLTL}} \vdash \varphi$; (4) $\Gamma^{\mathsf{infLTL}} \vDash \varphi$.

Proof. See Section 5.14.5. QED.

Therefore, infinite-trace LTL can be regarded as a theory containing two axioms, (INF) and (LIN), over the same signature Σ^{TS} for transition systems.

5.9.2 Defining finite-trace LTL

We have seen the syntax and semantics of finite-trace LTL in Section 2.9.2. Note that the finite-trace LTL syntax, namely the following

$$\varphi ::= p \in AP \mid \varphi \wedge \varphi \mid \neg\varphi \mid \circ\varphi \mid \varphi\, W\, \varphi$$

can be defined in matching μ-logic by introducing the following notation for W:

$$\text{“weak until”} \qquad \varphi_1\, W\, \varphi_2 \equiv \mu X \,.\, \varphi_2 \vee (\varphi_1 \wedge \circ X).$$

As for models, finite-trace LTL requires finite traces, so the underlying transition relations are linear and finite (i.e., there is no infinite trace). To capture both characteristics we add two axioms:

$$(\text{FIN})\ \ \mathsf{WF} \equiv \mu X \,.\, \circ X \qquad\qquad (\text{LIN})\ \ \bullet X \to \circ X$$

and call the resulting Σ^{TS}-theory Γ^{finLTL}. Intuitively, (FIN) forces all states to be well-founded, meaning that there is no infinite execution trace in the underlying transition systems.

Theorem 5.16. The following properties are equivalent for all finite-trace LTL formula φ: (1) $\vdash_{\mathsf{finLTL}} \varphi$; (2) $\vDash_{\mathsf{finLTL}} \varphi$; (3) $\Gamma^{\mathsf{finLTL}} \vdash \varphi$; (4) $\Gamma^{\mathsf{finLTL}} \vDash \varphi$.

Proof. See Section 5.14.6. QED.

Therefore, finite-trace LTL can be regarded as a theory containing two axioms, (Fin) and (Lin), over the same signature Σ^{TS} for transition systems.

5.9.3 Defining CTL

We have seen the syntax and semantics of CTL in Section 2.9.3. Note that the CTL syntax, namely the following

$$\varphi ::= p \in AP \mid \varphi \wedge \varphi \mid \neg\varphi \mid \mathsf{AX}\varphi \mid \mathsf{EX}\varphi \mid \varphi \,\mathsf{AU}\, \varphi \mid \varphi \,\mathsf{EU}\, \varphi$$

can be subsumed in matching μ-logic by introducing the following notations:

$$\mathsf{AX}\varphi \equiv \circ\varphi \qquad \varphi_1 \,\mathsf{AU}\, \varphi_2 \equiv \mu X.\, \varphi_2 \vee (\varphi_1 \wedge \circ X)$$
$$\mathsf{EX}\varphi \equiv \bullet\varphi \qquad \varphi_1 \,\mathsf{EU}\, \varphi_2 \equiv \mu X.\, \varphi_2 \vee (\varphi_1 \wedge \bullet X)$$

As for models, CTL requires infinite computation trees, so let us add the axiom (Inf) and call the resulting Σ^{TS}-theory Γ^{CTL}.

Theorem 5.17. For all CTL formulas φ, the following are equivalent: (1) $\vdash_{\mathsf{CTL}} \varphi$; (2) $\vDash_{\mathsf{CTL}} \varphi$; (3) $\Gamma^{\mathsf{CTL}} \vdash \varphi$; (4) $\Gamma^{\mathsf{CTL}} \vDash \varphi$.

Therefore, CTL can be regarded as a theory with one axiom, over the same signature Σ^{TS} for transition systems.

5.9.4 Discussion

It may be insightful to look at infinite-trace LTL, finite-trace LTL, CTL, as well as modal μ-calculus through the lenses of matching μ-logic, as theories over a unary symbol signature. Modal μ-calculus is the empty theory and thus the least constrained one. CTL comes immediately next with only one axiom, (Inf), to enforce infinite computation traces; Infinite-trace LTL further constrains with the linearity axiom (Lin). Finally, finite-trace LTL replaces (Inf) with (Fin). We believe that matching μ-logic can serve as a convenient and uniform framework to define and study temporal logics. For example, finite-trace CTL can be trivially obtained as the theory containing only the axiom (Fin). LTL with both finite and infinite traces is the theory containing only the axiom (Lin). And CTL with unrestricted (finite or infinite branch) models is the empty theory (i.e., modal μ-calculus).

5.10 DEFINING DYNAMIC LOGIC

We have seen the syntax of dynamic logic (DL) in Section 2.10. It is known that DL can be embedded in the variant of modal μ-calculus with multiple modalities (see, e.g., [65]). The idea is to define a modality $[a]$ for every atomic program $a \in APgm$, and then to define the four program constructs as least/greatest fixpoints. We can easily adopt the same approach and associate an empty matching μ-logic theory to DL, over a signature containing as many unary symbols as atomic programs. However, matching μ-logic allows us to propose a better embedding, unrestricted by the limitations of modal μ-calculus. Indeed, the embedding in [65] suffers from at least two limitations that we can avoid with matching μ-logic. First, sometimes transitions are not just labeled with discrete programs, such as in hybrid systems [66] and cyber-physical systems [67] where the labels are continuous values such as elapsing time. We cannot introduce for every time $t \in \mathbb{R}_{\geq 0}$ a modality $[t]$, as only countably many modalities are allowed. Instead, we may want to axiomatize the domains of such possibly continuous values and treat them as any other data. Second, we may want to quantify over such values, be they discrete or continuous, and we would not be able to do so (even in matching μ-logic) if they are encoded as modalities/symbols.

Let us instead define a signature for DL

$$\Sigma^{\mathsf{DL}} = (\{\mathsf{State}, \mathsf{Pgm}\}, \Sigma^{\mathsf{DL}}_{\epsilon,\mathsf{Pgm}} \cup \{\bullet \in \Sigma^{\mathsf{DL}}_{\mathsf{Pgm}\,\mathsf{State},\mathsf{State}}\})$$

where the "one-path next $\bullet$" is now a binary symbol taking an additional Pgm argument, and for all atomic programs $a \in APgm$ we add a constant symbol $a \in \Sigma^{\mathsf{DL}}_{\lambda,\mathsf{Pgm}}$. Just like how all Σ^{TS}-models are transition systems (Section 5.8), all Σ^{DL}-models are $APgm$-labeled transition systems. We define compound programs in DL as the following notations:

$$\begin{array}{ll}
\langle\alpha\rangle\varphi \equiv \bullet(\alpha,\varphi) & [\alpha]\varphi \equiv \neg\langle\alpha\rangle\neg\varphi \\
(\textsc{Seq}) \quad [\alpha\,;\beta]\varphi \equiv [\alpha][\beta]\varphi & (\textsc{Choice}) \quad [\alpha\cup\beta]\varphi \equiv [\alpha]\varphi \wedge [\beta]\varphi \\
(\textsc{Test}) \quad [\psi?]\varphi \equiv (\psi \to \varphi) & (\textsc{Iter}) \quad [\alpha^*]\varphi \equiv \nu X.\,(\varphi \wedge [\alpha]X)
\end{array}$$

Let Γ^{DL} denote the empty Σ^{DL}-theory.

Theorem 5.18. For all DL formulas φ, the following are equivalent: (1) $\vdash_{\mathsf{DL}} \varphi$; (2) $\vDash_{\mathsf{DL}} \varphi$; (3) $\Gamma^{\mathsf{DL}} \vdash \varphi$; (4) $\Gamma^{\mathsf{DL}} \vDash \varphi$.

Proof. See Section 5.14.8. QED.

We point out that the iterative operator $[\alpha^*]\varphi$ is axiomatized with two axioms in the proof system of DL in Figure 2.8:

$$\begin{aligned} &(\mathrm{DL}_6) \quad \varphi \wedge [\alpha][\alpha^*]\varphi \leftrightarrow [\alpha^*]\varphi \\ &(\mathrm{DL}_7) \quad \varphi \wedge [\alpha^*](\varphi \to [\alpha]\varphi) \to [\alpha^*]\varphi \end{aligned}$$

while we just regard it as notation, via (Iter). One may argue that (Iter) desugars to the operator ν, though, which obeys the proof rules (Pre-Fixpoint) and (Knaster Tarski) that essentially have the same effect as (DL_6) and (DL_7). We agree. And that is exactly why we think that we should have one uniform and fixed logic, such as matching μ-logic, where general fixpoint axioms are given to specify and reason about any fixpoint properties of any domains and to develop general-purpose automatic tools and provers. When it comes to specific domains and special-purpose logics, we can define them as theories/notations in MmL, as what we have done in this section for modal μ-calculus and all its fragment logics. Often, these special-purpose logics are simpler than matching μ-logic and more computationally efficient. In particular, modal μ-calculus and all its fragment logics shown in this section are not only complete but also decidable [68], while matching μ-logic does not have any complete proof system and thus its validity is not semi-decidable. Therefore, the existing decision procedures and completeness results of these special-purpose logics give decision procedures and completeness results (such as Theorem 5.14) for the corresponding matching μ-logic theories.

5.11 DEFINING REACHABILITY LOGIC

RL can be defined in matching μ-logic by defining the extended signature $\Sigma^{\mathsf{RL}} = \Sigma^{\mathsf{Cfg}} \cup \{\bullet \in \Sigma_{\mathsf{Cfg},\mathsf{Cfg}}\}$ and the following notation for reachability rules:

$$\begin{aligned} \text{“weak eventually”} \quad & \diamond_w \varphi \equiv \nu X.\, \varphi \vee \bullet X \quad // \text{ equal to } \neg \mathsf{WF} \vee \diamond\varphi \\ \text{“reaching star”} \quad & \varphi_1 \Rightarrow^* \varphi_2 \equiv \varphi_1 \to \diamond_w \varphi_2 \\ \text{“reaching plus”} \quad & \varphi_1 \Rightarrow^+ \varphi_2 \equiv \varphi_1 \to \bullet \diamond_w \varphi_2 \end{aligned}$$

Notice that the “weak eventually” $\diamond_w\varphi$ is defined similarly to the “eventually” $\diamond\varphi \equiv \mu X \,.\, \varphi \vee \bullet X$, but instead of using least fixpoint operator μ, we define it as a greatest fixpoint. One can prove that $\diamond_w\varphi = \neg\mathsf{WF} \vee \diamond\varphi$, that is, a configuration γ satisfies $\diamond_w\varphi$ if either it satisfies $\diamond\varphi$, or it is not well-founded, meaning that there exists an infinite execution path from γ. Also notice that “reaching plus” $\varphi_1 \Rightarrow^+ \varphi_2$ is a stronger version of “reaching star”, requiring that $\diamond_w\varphi_2$ should hold after at least one step. This progressive condition is crucial to the soundness of RL reasoning: as shown in (TRANSITIVITY) in Figure 2.12, circularities are flushed into the axiom set only after one reachability step is established. This leads us to the following translation from RL sequents to matching μ-logic patterns.

Definition 5.6. Given a rule $\varphi_1 \Rightarrow \varphi_2$, define the matching μ-logic pattern $\boxdot(\varphi_1 \Rightarrow \varphi_2) \equiv \Box(\varphi_1 \Rightarrow^+ \varphi_2)$ and extend it to a rule set A as follows: $\boxdot A \equiv \bigwedge_{\varphi_1 \Rightarrow \varphi_2 \in A} \boxdot(\varphi_1 \Rightarrow \varphi_2)$. Define the translation RL2MmL from RL sequents to matching μ-logic patterns as follows:

$$\text{RL2MmL}(A \vdash_C \varphi_1 \Rightarrow \varphi_2) = (\forall \boxdot A) \wedge (\forall \circ \boxdot C) \to (\varphi_1 \Rightarrow^\star \varphi_2)$$

where $\star = *$ if $C = \emptyset$ and $\star = +$ otherwise. We use $\forall \varphi$ as a shorthand for $\forall \vec{x} \,.\, \varphi$ where $\vec{x} = \mathit{freeVar}(\varphi)$. Recall that the "$\circ$" in $\forall \circ \boxdot C$ is "all-path next".

Hence, the translation of $A \vdash_C \varphi_1 \Rightarrow \varphi_2$ depends on whether C is empty or not. When C is nonempty, the RL sequent is stronger in that it requires at least one step being made in $\varphi_1 \Rightarrow \varphi_2$. Axioms (in A) are also stronger than circularities (in C) in that axioms always hold, while circularities only hold after at least one step because of the leading all-path next "$\circ$"; and since the "next" is an "all-path" one, it does not matter which step is actually made, as circularities hold on all next states.

Theorem 5.19. Let $\Gamma^{\mathsf{RL}} = \{\varphi \in \text{MLPATTERN}_{\mathsf{Cfg}} \mid M^{\mathsf{Cfg}} \vDash \varphi\}$ be the set patterns (without μ) of sort Cfg that hold in M^{Cfg}. For all RL systems S and rules $\varphi_1 \Rightarrow \varphi_2$ satisfying the same technical assumptions in [11], the following are equivalent: (1) $S \vdash_{\mathsf{RL}} \varphi_1 \Rightarrow \varphi_2$; (2) $S \vDash_{\mathsf{RL}} \varphi_1 \Rightarrow \varphi_2$; (3) $\Gamma^{\mathsf{RL}} \vdash \text{RL2MmL}(S \vdash_\emptyset \varphi_1 \Rightarrow \varphi_2)$; (4) $\Gamma^{\mathsf{RL}} \vDash \text{RL2MmL}(S \vdash_\emptyset \varphi_1 \Rightarrow \varphi_2)$.

Proof. See Section 5.14.9. QED.

Therefore, provided that an oracle for validity of all the configuration patterns in M^{Cfg} is available, the matching μ-logic proof system is capable of deriving any valid reachability rules. This way, matching μ-logic serves as an even more fundamental logic foundation than RL for the $\mathbb{K}$ framework (Section 2.15) and thus for programming language specification and verification, because matching μ-logic can express significantly more properties than partial correctness reachability.

5.12 DEFINING λ-CALCULUS

To define λ-calculus in matching μ-logic, we need to address two challenges. The first challenge is to handle the binding behavior of λ, that is, to define $\lambda x \,.\, e$ as a notation in matching μ-logic such that it satisfies the (meta-level) properties regarding free variables,

α-equivalence, and capture-avoiding substitution. The second challenge is to prove the following equivalent result, called the conservative extension theorem:

$$\Gamma^{\lambda} \vdash e_1 = e_2 \quad \xrightleftharpoons[\text{extensiveness}]{\text{conservativeness}} \quad \vdash_{\lambda} e_1 = e_2 \text{ for all } e_1, e_2 \in \Lambda \tag{5.8}$$

The conservativeness direction is the difficult part. Indeed, matching μ-logic has a richer syntax and a more complex proof system than λ-calculus. We need to show that this more complex infrastructure cannot be used to prove more equations between λ-expressions.

To solve the first challenge, we make an important observation that λ plays two important roles: (i) it builds a *term* $\lambda x \,.\, e$, and (ii) it builds a *binding* of x into e. We will separate these two roles when defining $\lambda x \,.\, e$ as a notation in matching μ-logic, where we build terms using symbols and creating the binding behavior using the built-in binder $\exists$.

To solve the second challenge, We give two different proofs for the conservativeness of Γ^{λ}, each providing a unique insight about the construction of Γ^{λ}. The first proof is a model-theoretic proof, discussed in Section 5.12.2. It considers the concrete cc models for λ-calculus, which are known to be complete with respect to λ-calculus reasoning (Section 2.11). This model-theoretic proof is easier to understand and is what inspired our encoding of the λ binder but it does not generalize to binders used in other formal systems, such as π-calculus or type systems. Therefore, we give another proof-theoretic proof, based purely on the syntax and formal proofs of λ-calculus, instead of its models. The proof-theoretic proof is easier to be generalized to the binders in other formal systems.

5.12.1 Defining the λ binder

Our definition of the λ binder in matching μ-logic is inspired by the concrete ccc models in Section 2.11. The key ingredient is the retraction function $\mathcal{G}$ that encodes representable functions into elements, so let us first define representable functions and the retraction function.

Recall that $f^{\rho}_{e,x}$ is the representable function in Section 2.11, which corresponds to the interpretation of $\lambda x \,.\, e$ under ρ in the concrete ccc model. The graph of $f^{\rho}_{e,x}$,

$$\mathsf{graph}(f^{\rho}_{e,x}) = \left\{ \left(a, |e|^{\lambda}_{\rho[a/x]}\right) \mid \text{for all } a \text{ in the concrete ccc model } A \right\} \tag{5.9}$$

contains all the argument-value pairs of $f^{\rho}_{e,x}$. Note that this graph is an element in $\mathcal{P}(A \times A)$, the powerset of $A \times A$, but not every element in $\mathcal{P}(A \times A)$ is the graph of a representable function. Therefore, the retraction function $\mathcal{G}$ is captured as a partial function from $\mathcal{P}(A \times A)$

to A which is defined only on the graphs of representable functions, and undefined elsewhere.

Now we start to define Γ^{λ} following the above intuition. Firstly, we include all λ-calculus variables in V as element (and not set) variables in matching μ-logic. Then, we define three sorts: Exp as the sort of λ-expressions; Pair as the product sort of V and Exp (Definition 5.2); and 2^{Pair} as its power sort (Definition 5.5). Intuitively, 2^{Pair} is the sort of all binary relations, including non-functions, over V and Exp, because the carrier set of 2^{Pair} is the powerset of the product of the carrier sets of V and Exp.

Next, we define a partial function $\mathsf{lambda}\colon 2^{\mathsf{Pair}} \rightharpoonup \mathsf{Exp}$, to represent the retraction function $\mathcal{G}$ in Section 2.11. We define $\mathsf{app}\colon \mathsf{Exp} \times \mathsf{Exp} \to \mathsf{Exp}$ to be the application function and write $e_1 e_2 \equiv \mathsf{app}(e_1, e_2)$. Abstraction $\lambda x \,.\, e$ is defined as the following syntactic sugar, where we extract the general binding notation $[x : V]\ e$ for clarity and because it can be used to define any other binders, not only λ:

$$[x : V]\ e \equiv \mathsf{intension}(\exists x : V \,.\, \langle x, e \rangle) \qquad // \text{ the binding notation} \qquad (5.10)$$

$$\lambda x \,.\, e \equiv \mathsf{lambda}([x : V]\ e) \qquad // \ \lambda\text{-abstraction} \qquad (5.11)$$

Equation (5.11) is a logical incarnation of the semantics of $\lambda x \,.\, e$ in the concrete ccc models into matching μ-logic. In a concrete ccc model, $|\lambda x \,.\, e|^{\lambda}_{\rho} = \mathcal{G}\left(f^{\rho}_{e,x}\right)$, where $f^{\rho}_{e,x}(a) = |e|^{\lambda}_{\rho[a/x]}$. In matching μ-logic, $\exists x : V \,.\, \langle x, e \rangle$ denotes the union set $\bigcup_x \{(x, e)\}$, namely the graph of $f^{\rho}_{e,x}$. Note that $\forall x : V \,.\, \langle x, e \rangle$ also yields the correct binding behavior, but it does not have the right semantic meaning of a graph. The binding notation $[x : V]\ e$ takes this graph as a *set* of pairs and *packs* them into one object in the power sort 2^{Pair}. Then, this packed object is passed to lambda, which decodes/retracts it into the intended interpretation of $\lambda x \,.\, e$. For now, we do not know any property about lambda, except that it is a partial function from 2^{Pair} to Exp. Its intended behavior will be axiomatized by the axiom schema (β)—the axiom schema that characterizes λ-abstraction and the semantics of λ.

Under the above notations, all λ-expressions are patterns. Particularly, the notation $\lambda x \,.\, e$ yields the right binding behaviors about λ via the built-in binder $\exists$. Let Γ^{λ} be the theory for λ-calculus that includes all the above definitions and notations and all instances of the (β) axiom schema:

$$\forall x_1 : V \,.\, \cdots \forall x_n : V \,.\, (\lambda x \,.\, e)\, e' = e[e'/x]$$

where $x_1, \ldots, x_n$ are all the free variables in $\mathit{freeVar}((\lambda x \,.\, e)\, e')$. Note that the axioms are needed to specify the semantics of λ in matching μ-logic, not its binding behavior. The latter is directly inherited from that of the built-in binder $\exists$.

We emphasize that the encoding of $\lambda x \,.\, e$ in Equations (5.10) and (5.11) is only possible

$$\begin{tikzcd}
\Gamma^\lambda \vdash e_1 = e_2 \arrow[r, Rightarrow, "1"'] & \Gamma^\lambda \vDash e_1 = e_2 \arrow[r, Rightarrow, "2"'] & M \vDash e_1 = e_2 \text{ for } \Sigma^\lambda\text{-models } M \arrow[d, Rightarrow, "3"] \\
\vdash_\lambda e_1 = e_2 & \vDash_\lambda e_1 = e_2 \arrow[l, Rightarrow, "5"'] & A \vDash_\lambda e_1 = e_2 \text{ for all concrete ccc models } A \arrow[l, Rightarrow, "4"']
\end{tikzcd}$$

Figure 5.1: Main Steps in the Model-Theoretic Conservativeness Proof

because matching μ-logic treats terms and formulas uniformly as patterns, and it allows (FOL-style) quantification to be built on terms. A similar definition will immediately fail in FOL, because FOL enforces a clear distinction between terms and formulas at the syntax level and quantification only applies to formulas.

We finish this section by proving the extensiveness theorem for λ-calculus.

Theorem 5.20. $\vdash_\lambda e_1 = e_2$ implies $\Gamma^\lambda \vdash e_1 = e_2$, for all $e_1, e_2 \in \Lambda$.

Proof. Note that Γ^λ contains all instances of (β) and equational reasoning is available in matching μ-logic. QED.

5.12.2 Model-theoretic conservativeness proof

Here we prove the conservativeness of Γ^λ using concrete ccc models of λ-calculus. The main proof steps are summarized in Figure 5.1. The only nontrivial one is Step 3, which requires to show that $M \vDash e_1 = e_2$ for all $M \vDash \Gamma^\lambda$ implies $A \vDash_\lambda e_1 = e_2$ for all A. The following is the key lemma that establishes the connection between concrete ccc models and Γ^λ-models.

Lemma 5.3. For any concrete ccc model A and any valuation ρ, there exists a Γ^λ-model M^A and a corresponding valuation ρ^A such that $|e|_{\rho^A} = \left\{|e|^\lambda_\rho\right\}$ for every $e \in \Lambda$.

Proof. Let us fix a concrete ccc model $(A, \_\bullet_A\_, \mathcal{G})$, where $R(A)$ is its set of representable functions and $\mathcal{G}\colon R(A) \to A$ is its retraction function. Let $M^A_{\mathsf{Exp}} = A$. By Proposition 5.1 and **??**, $M^A_{\mathsf{Pair}} = A \times A$ and $M^A_{2^{\mathsf{Pair}}} = \mathcal{P}(A \times A)$. We define lambda_{M^A} accordingly to the retraction function $\mathcal{G}$; i.e., $\mathsf{lambda}_{M^A}(P) = \{\mathcal{G}(f)\}$ whenever $P = \mathsf{graph}(f)$ and $f \in R(A)$, and $\mathsf{lambda}_{M^A}(P) = \emptyset$, otherwise.

We define the corresponding ρ^A as $\rho^A(x) = \rho(x)$ for every $x \in V$. We prove that $|e|_{\rho^A} = \{|e|^\lambda_\rho\}$ for every $e \in \Lambda$ by structural induction on e. The only nontrivial case is when e is $\lambda x \,.\, e_1$. In this case, we have

$$\begin{aligned}
|\lambda x \,.\, e_1|_{\rho^A} &= |\mathsf{lambda}\,(\mathsf{intension}\,(\exists x \colon V \,.\, \langle x, e_1\rangle))|_{\rho^A} \\
&= \mathsf{lambda}_{M^A}(|\mathsf{intension}\,(\exists x \colon V \,.\, \langle x, e_1\rangle)|_{\rho^A})
\end{aligned}$$

$$
\begin{aligned}
&= \mathsf{lambda}_{M^A}(|\exists x : V \,.\, \langle x, e_1\rangle)|_{\rho^A})\\
&= \mathsf{lambda}_{M^A}(\bigcup_{a\in A}\{(a, |e_1|_{\rho^A[a/x]})\})\\
&= \mathsf{lambda}_{M^A}(\bigcup_{a\in A}\{(a, |e_1|^\lambda_{\rho[a/x]})\})\\
&= \mathsf{lambda}_{M^A}(\mathsf{graph}(f^\rho_{e_1,x}))\\
&= \{\mathcal{G}(f^\rho_{e_1,x})\}\\
&= \{|\lambda x \,.\, e_1|^\lambda_\rho\}
\end{aligned}
$$

Finally, we need to verify that $M^A \vDash (\beta)$. It is straightforward. Using the above result, for any $x \in V$, $e, e' \in \Lambda$, and ρ, we have that $|(\lambda x \,.\, e)e'|^\lambda_\rho = |e[e'/x]|^\lambda_\rho$ in A implies $|(\lambda x \,.\, e)e'|_{\rho^A} = |e[e'/x]|_{\rho^A}$ in M^A. Noting that ρ^A is arbitrary (as ρ is arbitrary), $M^A \vDash (\beta)$. QED.

The operations $\mathsf{intension}$ and lambda are crucial in the proof of Lemma 5.3. Without them, $\exists x : V \,.\, \langle x, e\rangle$ is merely the graph set, not a singleton pattern, and thus cannot be directly used to interpret $\lambda x \,.\, e$.

Using Lemma 5.3, we can immediately prove Step 3 in Figure 5.1:

Lemma 5.4. If $M \vDash e_1 = e_2$ for every Γ^λ-model M, then $A \vDash_\lambda e_1 = e_2$ for every concrete ccc models A.

Proof. Let A be any concrete ccc model and ρ be any valuation. By Lemma 5.3, there exists a Γ^λ-model M^A and a valuation ρ^A such that $|e|_{\rho^A} = \{|e|^\lambda_\rho\}$ for any $e \in \Lambda$. Since $M^A \vDash e_1 = e_2$, we have $|e_1|_{\rho^A} = |e_2|_{\rho^A}$, and thus $|e_1|^\lambda_\rho = |e_2|^\lambda_\rho$. Since ρ is arbitrary, $A \vDash_\lambda e_1 = e_2$. QED.

Now we finish the model-theoretic conservativeness proof in Figure 5.1.

Theorem 5.21. $\Gamma^\lambda \vdash e_1 = e_2$ implies $\vdash_\lambda e_1 = e_2$, for all $e_1, e_2 \in \Lambda$.

Proof. See Figure 5.1, where Step 1 is proved by Theorem 4.1; Step 2 is proved by definition; Step 3 is proved by Lemma 5.4; Step 4 is proved by definition; and Step 5 is proved by Theorem 2.13. QED.

Theorem 5.21 together with Theorem 5.20 show that Γ^λ is a conservative extension of λ-calculus.

Theorem 5.22. For every $e_1, e_2 \in \Lambda$, these are equivalent: (1) $\Gamma^\lambda \vdash e_1 = e_2$; (2) $\Gamma^\lambda \vDash e_1 = e_2$; (3) $\vDash_\lambda e_1 = e_2$; (4) $\vdash_\lambda e_1 = e_2$.

Proof. (1) $\Longrightarrow$ (2) is by Theorem 4.1. (2) $\Longrightarrow$ (3) is by Lemma 5.4. (3) $\Longrightarrow$ (4) is by Lemma 2.13. (4) $\Longrightarrow$ (1) is by Theorem 5.21. QED.

The equivalence (1) $\iff$ (4) is called the conservative extension theorem for Γ^λ. The equivalence (2) $\iff$ (4) is called the (deductive) completeness of matching μ-logic with respect to Γ^λ. By defining λ-calculus in matching μ-logic, we automatically obtain a model of theory for λ-calculus via the matching μ-logic Γ^λ-models.

5.12.3 Proof-theoretic conservativeness proof

The model-theoretic conservativeness proof is intuitive because it is based on the models of λ-calculus. It also has a clear limitation, which is that it requires a model theory for λ-calculus. In Section 5.12.2, we use the concrete ccc models for λ-calculus and especially the completeness result in Theorem 2.13. Therefore, the model-theoretic proof in Section 5.12.2 is specific to λ-calculus and the concrete ccc models. It is not easy to generalize to the binders in other formal systems, especially those do not have an accessible model theory.

Therefore, we give an an alternative, proof-theoretic conservativeness proof that is entirely based on the syntactic structure of λ-calculus. As a result, the proof-theoretic proof is easier to generalize to other logical systems and binders.

We build a special Γ^λ-model T, which we call the *term model* of λ-calculus,[2] and follow the term algebra technique [69, 70, 71]: T has as elements the equivalence classes of λ-expressions modulo $\alpha\beta$-equivalence, and each $e \in \Lambda$ is interpreted in T as the equivalence class containing itself, denoted by $[e]$. Formally, we will prove this:

Theorem 5.23. Let $[e] = \{e' \in \Lambda \mid \ \vdash_\lambda e = e'\}$ be the equivalence class of e modulo $\alpha\beta$-equivalence. Let $[\Lambda] = \{[e] \mid e \in \Lambda\}$ be the set of all these classes. Then, there is a Γ^λ-model T, called *term model*, and a valuation ρ_T, called *term valuation*, such that $|e|_{T,\rho_T} = \{[e]\}$ for all $e \in \Lambda$. Since T is a fixed model, we abbreviate $|e|_{T,\rho_T}$ as $|e|_{\rho_T}$.

Note that for distinct variables $x, y \in V$, we have $[x] \neq [y]$ [29, Fact 2.1.37]. Clearly, $x \in [x]$, but $[x]$ also includes infinitely many expressions: $(\lambda y\,.\,y)x$, $(\lambda y\,.\,y)((\lambda y\,.\,y)x)$, etc.

We will show the construction of T shortly after. For now, let us first prove Theorem 5.21 from Theorem 5.23.

Another Proof of Theorem 5.21. Suppose $\Gamma^\lambda \vdash e_1 = e_2$. We have

$$\begin{aligned} \Gamma^\lambda \vdash e_1 = e_2 \quad &\Rightarrow \quad \Gamma^\lambda \vDash e_1 = e_2 && \text{by Theorem 4.1} \\ &\Rightarrow \quad T \vDash e_1 = e_2 && \text{by definition} \end{aligned}$$

[2] In the literature on λ-calculus, term models have a different meaning. For example, in [29], term models are special λ-calculus models constructed based on the combinatory algebra semantics; see Section 5.12.4 for a comparison.

$$
\begin{aligned}
&\Rightarrow\quad |e_1|_{\rho_T} = |e_2|_{\rho_T} && \text{by Proposition 2.16}\\
&\Rightarrow\quad [e_1] = [e_2] && \text{by Theorem 5.23}\\
&\Rightarrow\quad \vdash_\lambda e_1 = e_2 && \text{by the definition of } [e] \text{ in Theorem 5.23}
\end{aligned}
$$

QED.

Now we construct T and show that $T \vDash \Gamma^\lambda$. We define $T_{\mathsf{Exp}} = [\Lambda]$, which is the set of equivalence classes of λ-expressions. Note that $T_{\mathsf{Pair}} = [\Lambda] \times [\Lambda]$ and $T_{2^{\mathsf{Pair}}} = \mathcal{P}([\Lambda] \times [\Lambda])$. We define $\mathsf{app}_T([e_1], [e_2]) = [e_1\, e_2]$ for $e_1, e_2 \in \Lambda$. Note that this definition is well-defined, because $\vdash_\lambda e_1\, e_2 = e_1'\, e_2'$ whenever $\vdash_\lambda e_1 = e_1'$ and $\vdash_\lambda e_2 = e_2'$. Finally, we define

$$\mathsf{lambda}_T\left(\bigcup_{z \in V} ([z], [e[z/x]])\right) = \big\{[\lambda x\,.\,e]\big\}, \quad \text{for any } x \in V \text{ and } e \in \Lambda. \tag{5.12}$$

and $\mathsf{lambda}_T(P) = \emptyset$, if P is not a graph of the above form.

The construction of T, especially Equation (5.12), is critically depending on the notation definition $\lambda x\,.\,e \equiv \mathsf{lambda}\,(\mathsf{intension}\,\exists x : V\,.\,\langle x, e\rangle)$. The α-equivalence $\lambda x\,.\,e \equiv \lambda z\,.\,(e[z/x])$ is captured, both syntactically and semantically, by collecting all the pairs $\langle z, e[z/x]\rangle$ for all z, using the pattern $\exists x : V\,.\,\langle x, e\rangle$. Therefore, $\exists x : V\,.\,\langle x, e\rangle$ encapsulates all the information about $[\lambda x\,.\,e]$, which is *packed* by $\mathsf{intension}$ and passed to lambda, and then *retracted* to restore the original expression $\lambda x\,.\,e$. Proposition 5.24 shows that the condition in Equation (5.12) on lambda_T is consistent.

Proposition 5.24. $[\lambda x\,.\,e] = [\lambda x'\,.\,e']$, *whenever*

$$\bigcup_{z \in V} ([z], [e[z/x]]) = \bigcup_{z \in V} ([z], [e'[z/x']]) \tag{5.13}$$

Proof. Assume the opposite, i.e., $[\lambda x\,.\,e] \neq [\lambda x'\,.\,e']$. Let $z^* \in V$ be a fresh variable that does not occur in $\lambda x\,.\,e$ or $\lambda x'\,.\,e'$. Then we have $\lambda x\,.\,e \equiv \lambda z^*\,.\,e[z^*/x]$ and $\lambda x'\,.\,e' \equiv \lambda z^*\,.\,e'[z^*/x']$. By the assumption, we have $[\lambda z^*\,.\,e[z^*/x]] \neq [\lambda z^*\,.\,e'[z^*/x']]$, and thus $[e[z^*/x]] \neq [e'[z^*/x']]$. Noting that $[z_1] = [z_2]$ iff $z_1 = z_2$, for every $z_1, z_2 \in V$. Thus, $([z^*], [e[z^*/x]])$ is in the left-hand side of Equation (5.13) but not the right-hand side. This is a contradiction. QED.

So far, we have constructed the term model T. We now define the term valuation ρ_T as $\rho_T(x) = [x]$ for every $x \in V$.

Proposition 5.25. $|e|_{\rho_T} = \{[e]\}$, *and* $|e|_{\rho[\rho(z)/x]} = |e[z/x]|_\rho$ *for all* ρ.

Proof. We prove both properties simultaneously by induction on the λ-*depth* $d(e)$ of e, which is the maximum number of nested λ's in e. If $d(e) = 0$ then e is a variable or is built purely using applications (i.e., app) and has no λ abstraction. In this case, both properties can be proved by another structural induction on e. If $d(e) \geq 1$ then e has either the form $e_1\, e_2$ where $d(e_1), d(e_2) \leq d(e)$, or the form $\lambda x \,.\, e_1$ where $d(e_1) \leq d(e) - 1$. Then another structural induction on e proves both properties. QED.

Proposition 5.26. *If* $\vdash_\lambda e = e'$, *then* $|e|_\rho = |e'|_\rho$ *for any* $\rho \in \mathit{VarVal}$.

Proof. Note that the interpretation of a λ-expression relies on its free variables. Suppose $\mathit{freeVar}(e) \cup \mathit{freeVar}(e') = \{x_1, \ldots, x_n\}$ and $\rho(x_i) = [y_i]$ for $i \in \{1, \ldots, n\}$. Then, y_i is the only variable that is in $[y_i]$. Since ρ equals to $\rho_T[[y_1]/x_1] \cdots [[y_n]/x_n]$ restricted on $x_1, \ldots, x_n$, we have $|e|_\rho = |e|_{\rho_T[[y_1]/x_1]\cdots[[y_n]/x_n]}$. By Proposition 5.25, $|e|_{\rho_T[[y_1]/x_1]\cdots[[y_n]/x_n]} = |e[y_1/x_1] \cdots [y_n/x_n]|_{\rho_T} = \{[e[y_1/x_1] \cdots [y_n/x_n]]\}$. Similarly $|e'|_\rho = \{[e'[y_1/x_1] \cdots [y_n/x_n]]\}$. Then, $\vdash_\lambda e[y_1/x_1] \cdots [y_n/x_n] = e'[y_1/x_1] \cdots [y_n/x_n]$. Then we have

$$[e[y_1/x_1] \cdots [y_n/x_n]] = [e'[y_1/x_1] \cdots [y_n/x_n]]$$

Hence, $|e|_\rho = |e'|_\rho$. QED.

Now we only need to prove Theorem 5.23.

Proof of Theorem 5.23. We have shown that $|e|_{\rho_T} = \{[e]\}$ for every $e \in \Lambda$, in Proposition 5.25. It remains to show that T validates (β), i.e., $|(\lambda x \,.\, e)\, e'|_\rho = |e[e'/x]|_\rho$ for all ρ. The latter follows immediately from Proposition 5.26. QED.

5.12.4 Discussion

We first compare our term model T for λ-calculus to the other classic notion of term models. In λ-calculus, a *term model* [29, Definition 5.2.11] is a special λ-model, which is an algebra with $[\Lambda]$ being the underlying carrier set. The operations include a binary application function given by $[e_1][e_2] = [e_1 e_2]$ for $e_1, e_2 \in \Lambda$ as well as two constants: $k = [\lambda x \,.\, \lambda y \,.\, x]$ and $s = [\lambda x \,.\, \lambda y \,.\, \lambda z \,.\, (xz)(yz)]$. We denote the above model by A and call it a *classical term model*, to not confuse it with our term model T. Clearly, T and A follow different approaches to capture λ-expressions. While A uses the name-free, combinators approach, where λ is handled by abstraction elimination, our term model T gives an explicit and constructive interpretation to λ, as shown in Equation (5.12).

Next, we discuss the *representability problem* [72, pp. 8], which is a long-standing, concerning and open problem in the study of λ-calculus. The problem asks if a given class of λ-calculus models is *representationally complete*, in the sense that there exists a model in the given class such that any two expressions e_1 and e_2 are provably equal if and only if they are interpreted as the same element/value in that model. Representability completeness indicates that a class of λ-calculus models is sufficient in capturing the formal reasoning in λ-calculus, so one may reduce the study of formal reasoning in λ-calculus to the study of models, where more mathematical tools and techniques can be applied. Hence, reduction is the main motivation.

λ-calculus models are broadly divided into *syntactic models* and *non-syntactic models* [73, pp. 13], depending on whether their construction is based on the syntax and provability of λ-calculus or not. All the classical term models in λ-calculus, as well as our particular term model in Section 5.12.3, are syntactic models. Syntactic models are often representationally complete, but studying them tends to be as hard as studying the syntax and formal reasoning directly, and thus the reduction to syntactic models usually does not help simplify the study of λ-calculus. Thus, for decades researchers have been searching for and studying sub-classes of non-syntactic concrete ccc models, hoping they are also representationally complete. So far, three main such sub-classes have been identified, known as the *main semantics* of λ-calculus: Scott's continuous semantics [74], Berry's stable semantics [75, 76], and Bucciarelli-Ehrhard strongly stable semantics [77]. The representability problem for the main semantics (and their sub-classes) has remained largely open as of today, except for some negative results proved for some sub-classes (e.g., graph models [78]).

Theorem 5.23 shows that the class of Γ^λ-models of is representationally complete, positively answering the representability problem for our matching μ-logic semantics of λ-calculus. Our proof does not rely on any known results about the representational completeness of any existing semantics; instead, it is entirely based on the model theory of matching μ-logic, which is not specific to λ-calculus but which allows for an appropriate axiomatization of λ-calculus as a theory that is hereby endowed with the desired representationally complete models automatically. We can push Theorem 5.23 even further to any equational extensions of λ-calculus, known as λ-*theories*. Indeed, the definition of the equivalence class $[e]$ as the set of $\alpha\beta$-equivalent expressions of e, has not been critical in the proof of Theorem 5.23, and the conclusion still holds if we consider any equivalence class $[e]$ that includes the basic $\alpha\beta$-equivalence. Therefore, we conclude that the matching μ-logic definition of λ-calculus is representationally complete for all λ-theories.

Although we do not solve any of the existing open problems, our work suggests the matching μ-logic can be a viable alternative to the existing λ-calculus models within the main semantics. The matching μ-logic models are as good as the existing models for λ-calculus in terms of theoretical properties w.r.t. formal reasoning and semantics, yet unlike the existing models, they are general in the sense that they are not crafted specifically for λ-calculus, but are obtained from the matching μ-logic theory Γ^{λ}. We give a general solution for all the binders, which for λ-calculus is as good as the state of the art, considering both the proof-theoretic and the model-theoretic aspects.

5.13 DEFINING TERM-GENERIC LOGIC

We have shown how to define the λ binder as the following notation (Eqs. (5.10)-(5.11)):

$$\lambda x \,.\, e \equiv \mathsf{lambda}\, [x : V]\, e \tag{5.14}$$

In this section we show that our approach is not specific to λ-calculus. We provide evidence that matching μ-logic can serve as a general approach to dealing with binders. We will show how to use patterns similar to Eq. (5.14) to define the binders in a variety of logical systems, including System F [79, 80], pure type systems [81], π-calculus [82], and more, and prove a corresponding conservative extension theorem for each of them. To do that, several challenges need to be solved.

The first challenge is that binders can have more complex binding behavior than in λ-calculus; see Figure 5.2. For example, $\lambda x : e_1 \,.\, e_2$ in System F binds x within e_2, but not in e_1; $\mathsf{Inp}(x, y, e)$ in π-calculus has the binding variable in the second position (i.e., y), and not the first position. We deal with this binding behavior by desugaring to binders whose binding variable is their first argument and is bound within the second argument only; that is, we desugar an arbitrary binder to a binder of the form $b(x, e_1, \ldots, e_n)$, where x is bound in e_1 but not in $e_2, \ldots, e_n$. Clearly, this desugaring process is just a sequence of argument swappings. Then, we further desugar $b(x, e_1, \ldots, e_n)$ to $b'(b''(x, e_1), e_2, \ldots, e_n)$, where b' is a (binding-free) symbol and b'' is a binder that binds x to e_1, just like λ in λ-calculus. Finally, we define $b''(x, e_1)$ as the following syntactic sugar:

$$b''(x, e) \equiv \mathsf{retraction}_b\, [x : V]\, e \tag{5.15}$$

in the same way as in Eq. (5.14), except that here we use a new retraction symbol $\mathsf{retraction}_b$ that is specific to the binder b. Each binder has its own retraction symbol, but the other infrastructure symbols, such as products, powersets, and the binding notation $[x : V]\, e$, are the same. From now on, we will only consider binders $b(x, e)$ that bind x within e, for technical convenience.

The second challenge is that logical systems featuring bindings are very different from each other, in terms of the kinds of logical reasoning that is carried out in them. For example,

Binders	**Behaviors**	**Meaning**	**Systems**
$\lambda x \,.\, e$	binding x into e	function abstraction	λ-calculus
$\lambda x : e_1 \,.\, e_2$	binding x into e_2	function abstraction	System F
$\lambda t \,.\, e$	binding t into e	type abstraction	System F
$\Pi t \,.\, e$	binding t into e	Π-type constructor	System F
$\lambda x : e_1 \,.\, e_2$	binding x into e_2	function abstraction	Pure type system
$\pi x : e_1 \,.\, e_2$	binding x into e_2	type abstraction	Pure type system
$\mathsf{Inp}(x, y, e)$	binding y into e	input process	π-calculus
$\nu y \,.\, e$	binding y into e	new process name creation	π-calculus
$\mathsf{Bout}(e_1, x, y, e_2)$	binding y into e_2	bound output transition	π-calculus
$\mathsf{Inp}(e_1, x, y, e_2)$	binding y into e_2	input transition	π-calculus

Figure 5.2: Example Binders and Their Behavior in Logical Systems

System F derives *typing judgments* $\Gamma \rhd e_1 : e_2$ to mean that e_1 has type e_2 under typing environment Γ; π-calculus derives *transitions* $e_1 \xrightarrow{act} e_2$ to mean that process e_1 transits by action *act* to process e_2. It is tedious and non-systematic to consider these logical systems separately, because we would need to capture their specific logical reasoning and prove the conservative extension theorem for each of them, more or less similarly to the syntax-based proof in Section 5.12.3.

To capture the various logical systems featuring bindings more systematically, we employ a parametric framework for binders, called *term-generic logic* [30] (TGL), discussed in Section 2.12. We will define TGL in matching μ-logic and prove a conservative extension theorem for TGL, from which the conservative extension theorems for the other logical systems follow as corollaries. We define a theory Γ^{TGL} and introduce notations such that all TGL terms and formulas are well-formed patterns. We show that Γ^{TGL} is a conservative extension of TGL, by proving the following equivalence theorem.

Theorem 5.27. Under the condition in Theorem 2.14, the following are equivalent: (1) $(\Gamma^{\mathsf{TGL}} \cup E) \vdash \bigwedge \Delta_1 \to \bigvee \Delta_2$. (2) $(\Gamma^{\mathsf{TGL}} \cup E) \vDash \bigwedge \Delta_1 \to \bigvee \Delta_2$; (3) $E \vdash_{\mathsf{TGL}} \Delta_1 \rhd \Delta_2$; (4) $E \vdash_{\mathsf{TGL}} \Delta_1 \rhd \Delta_2$; Here, $\bigwedge \Delta_1$ is the conjunction of patterns in Δ_1 and $\bigvee \Delta_2$ is the disjunction of patterns in Δ_2.

The many-sorted binder syntax and TGL terms are captured by defining sorts and many-sorted functions, and defining binders as in Eq. (5.15). TGL formulas, except $\pi(e_1, \ldots, e_n)$, are captured by matching μ-logic's derived connectives and equality. Predicate $\pi(e_1, \ldots, e_n)$ for $\pi \in \Pi_{s_1 \cdots s_n}$, is captured by defining a symbol π and the following axiom:

$$(\textsc{Predicate}) \quad \forall x_1 : s_1 \,.\, \ldots \forall x_n : s_n \,.\, (\pi\, x_1 \, \cdots \, x_n = \top) \vee (\pi\, x_1 \, \cdots \, x_n = \bot) \qquad (5.16)$$

which specifies that π returns either $\top$ or $\bot$, i.e., it indeed builds predicate patterns. Without such axioms, $\pi\, x_1 \cdots x_n$ could be any subset. Let Γ^{TGL} contain all the above definitions and notations. This way, all TGL terms are functional patterns and all TGL formulas are predicate patterns.

Theorem 5.27 is proved using a model-based approach similar to Figure 5.1. Here we explain the only nontrivial proof step, which is (2) $\Longrightarrow$ (3). This is proved by constructing a matching μ-logic model M^A from any given TGL model A, such that all TGL terms and formulas are interpreted the same in M^A and A, i.e., $|e|_\rho = \{A_e(\rho)\}$ for every $e \in \mathrm{TGL}\textsc{Term}$; $|\varphi|_\rho = M^A$ whenever $\rho \in A_\varphi$, and $|\varphi|_\rho = \emptyset$, whenever $\rho \notin A_\varphi$, for every $\varphi \in \mathrm{TGL}\textsc{Form}$.

Using TGL and Theorem 5.27, we obtain a systematic proof of the conservative extension theorems and deductive completeness theorems for all logical systems that have been defined in TGL and studied in [30, Section 4] and [83, Section 4], including System F [79, 80] (both the typing and reduction versions), λ-calculus (including the untyped [28], sub-typed [84], illative [29], and linear versions [85, 86]), pure type systems [81], and π-calculus [82]. The systematic proof works as follows. For each logical system L, its set of terms Term_L can be captured by a binder syntax using the desugaring discussed at the beginning of Section 2.12. The proof/type system of L that derives sequents of the form $\vdash_L \Phi$ is captured by a set of TGL axioms E^L, where each axiom corresponds to one type/proof rule of L [30]. An *adequacy theorem* is also proved there for each L, stating that $\vdash_L \Phi$ iff $E^L \vdash_{\mathsf{TGL}} \Phi^{\mathsf{TGL}}$, where Φ^{TGL} (of the form $\Delta_1^\Phi \rhd \Delta_2^\Phi$) is the corresponding TGL encoding of the L-sequent Φ. Let $\Gamma^L = \Gamma^{\mathsf{TGL}} \cup E^L$ be the theory that captures L, and $\Phi^{\mathsf{ML}} = \bigwedge \Delta_1^\Phi \to \bigvee \Delta_2^\Phi$ be the encoding of Φ. By Theorem 2.14, we have that $\vdash_L \Phi$ in L, iff $E^L \vdash_{\mathsf{TGL}} \Phi^{\mathsf{TGL}}$ in TGL, iff $\Gamma^L \vdash \Phi^{\mathsf{ML}}$ in matching μ-logic, iff $\Gamma^L \vDash \Phi^{\mathsf{ML}}$ in matching μ-logic. Hence, Γ^L is a conservative extension of L and the class of matching logic models of Γ^L is complete with respect to L.

Note that the term *consistency* has different meanings in different contexts. In type systems, inconsistency means the ability to prove any typing judgments $t : \tau$. Similarly, in λ-calculus or other equational logic theories, inconsistency means the ability to prove any equations $e_1 = e_2$. However, in matching μ-logic (and also FOL), inconsistency means the ability to prove logical false $\bot$. Thus, inconsistency for classical logics such as matching μ-logic is stricter than that for type systems and λ-calculus. For example, if T is a PTS that contains the typing axiom $\mathit{Type} : \mathit{Type}$, then T is inconsistent [87], but Γ^T is still a consistent matching μ-logic theory and has a model that interprets the typing relation $\_ : \_$ as the total relation on all PTS terms.

5.14 PROOFS

We present proof details for the results in Chapter 5.

5.14.1 Proof of Theorem 5.2

Even though we tacitly blur the distinction between constant symbol $\sigma \in \Sigma_{\lambda, s_1 \otimes \cdots \otimes s_n \otimes s}$ and n-ary symbol $\sigma \in \Sigma_{s_1 \ldots s_n, s}$, doing so will cause us a lot of trouble in this section, when our aim is to prove such a blur of syntax actually works. Therefore, within this section, we introduce and use a more distinct syntax that distinguishes the two, as follows

$\sigma \in \Sigma_{s_1, \ldots, s_n, s}$	an n-ary symbol
$\alpha_\sigma \in \Sigma_{\lambda, s_1 \otimes \cdots \otimes s_n \otimes s}$	the corresponding constant symbol
$\sigma(\varphi_1, \ldots, \varphi_n)$	symbol application
$\alpha_\sigma[\varphi_1, \ldots, \varphi_n]$	projections
$\sigma(x_1, \ldots, x_n) = \alpha_\sigma[x_1, \ldots, x_n]$	recursive symbol
$\alpha_\sigma = \mu\alpha \,.\, \exists\vec{x}\langle\vec{x}, \varphi[\alpha/\sigma]\rangle$	definition of α_σ

Before we prove Theorem 5.2, we introduce a useful lemma that allows us to prove properties about least fixpoint patterns. Recall that rule (Knaster Tarski) allows us to prove theorems of the form $\Gamma \vdash \mu X \,.\, \varphi \to \psi$. However, in practice, the least fixpoint pattern $\mu X \,.\, \varphi$ is not always the only components on the left hand side, but rather stay *within some contexts*. The following lemma is particularly useful in practice, as it allows us to "plug out" the least fixpoint pattern from its context, so that we can apply (Knaster Tarski). After that, we may "plug it back" into the context.

Lemma 5.5. Let $C[\square]$ be a context such that $\square$ does not occur under any μ's, and

- $C[\varphi \wedge \psi] = C[\varphi] \wedge \psi$, for all patterns φ and all predicate patterns ψ;
- $C[\exists x \,.\, \varphi] = \exists x \,.\, C[\varphi]$, for all φ and $x \notin \mathit{freeVar}(C[\square])$.

Then we have that $\Gamma \vdash C[\varphi] \to \psi$ if and only if $\Gamma \vdash \varphi \to \exists x \,.\, x \wedge \lfloor C[x] \to \psi \rfloor$.

Proof. We prove both directions simultaneously. Note that it is easy to prove that $\Gamma \vdash \varphi = \exists x \,.\, (x \wedge (x \in \varphi))$ using rules (Membership) in the proof system $\mathcal{P}$ (see Figure 2.11).

We start with $\Gamma \vdash C[\varphi] \to \psi$. By the mentioned equality, we get $\Gamma \vdash C[\exists x \,.\, (x \wedge (x \in \varphi))] \to \psi$. By the properties of C, it becomes $\Gamma \vdash (\exists x \,.\, C[x] \wedge x \in \varphi) \to \psi$, which, by FOL reasoning, becomes $\Gamma \vdash x \in \varphi \to (C[x] \to \psi)$. Note that $x \in \varphi$ is a predicate pattern, so the goal is equivalent to $\Gamma \vdash x \in \varphi \to \lfloor C[x] \to \psi \rfloor$.

Now we are almost done. To show the "if" part, we apply (Membership Introduction) on $\Gamma \vdash \varphi \to \exists x \,.\, x \wedge \lfloor C[x] \to \psi \rfloor$ and obtain $\Gamma \vdash y \in \varphi \to \exists x \,.\, (y \in x) \wedge \lfloor C[x] \to \psi \rfloor$. Note that y is a fresh variable and $y \notin \mathit{freeVar}(C[x]) \cup \mathit{freeVar}(\psi)$, so $y \in \lfloor C[x] \to \psi \rfloor = \lfloor C[x] \to \psi \rfloor$. Notice that $y \in x = (y = x)$. And we obtain $\Gamma \vdash y \in \varphi \to \lfloor C[y] \to \psi \rfloor$. Done.

To show the "only if" part, we apply some simple FOL reasoning on $\Gamma \vdash x \in \varphi \to \lfloor C[x] \to \psi \rfloor$ and conclude that $\Gamma \vdash (\exists x \,.\, (x \wedge x \in \varphi)) \to \exists x \,.\, (x \wedge \lfloor C[x] \to \psi \rfloor)$. Then by the equality $\varphi = \exists x \,.\, (x \wedge x \in \varphi)$, we are done. QED.

Note the conditions about the context C in Lemma 5.5 are important. Many contexts that arise in practice satisfy the conditions. In particular, (nested) symbol contexts satisfy the conditions automatically.

Under the above new notation and the lemma, we are ready to prove Theorem 5.2.

Proof of Theorem 5.2. (Pre-Fixpoint). This is proved by simply unfolding α_σ following its definition.

(Knaster Tarski). We give the following proof that goes backward from conclusion to their sufficient conditions.

$$\begin{aligned}
& \sigma(x_1, \ldots, x_n) \to \psi \\
\Longleftarrow \quad & \alpha_\sigma[x_1, \ldots, x_n] \to \psi \\
\Longleftarrow \quad & \alpha \to \exists \alpha \,.\, (\alpha \wedge \lfloor \alpha[x_1, \ldots, x_n] \to \psi \rfloor) \\
\Longleftarrow \quad & \alpha_\sigma \to \forall \vec{x} \,.\, \underbrace{\exists \alpha \,.\, (\alpha \wedge \lfloor \alpha[x_1, \ldots, x_n] \to \psi \rfloor)}_{\alpha_0} \\
\Longleftarrow \quad & \exists \vec{x} \,.\, \langle \vec{x}, \varphi[\forall \vec{x} \,.\, \alpha_0 / \sigma] \rangle \to \forall \vec{x} \,.\, \alpha_0 \\
\Longleftarrow \quad & \langle \vec{x}, \varphi[\forall \vec{x} \,.\, \alpha_0 / \sigma] \rangle \to \alpha_0[z_1/x_1 \ldots z_n/x_n] \\
\Longleftarrow \quad & \langle \vec{x}, \varphi[\forall \vec{x} \,.\, \alpha_0 / \sigma] \rangle \\
& \quad \to \exists \alpha \,.\, (\alpha \wedge \lfloor \alpha[z_1, \ldots, z_n] \to \psi[z_1/x_1 \ldots z_n/x_n] \rfloor) \\
\Longleftarrow \quad & \langle \vec{x}, \varphi[\forall \vec{x} \,.\, \alpha_0 / \sigma] \rangle [x_1, \ldots, x_n] \to \psi \\
\Longleftarrow \quad & \varphi[\forall \vec{x} \,.\, \alpha_0 / \sigma] \to \psi \\
\Longleftarrow \quad & \varphi[\forall \vec{x} \,.\, \alpha_0 / \sigma] \to \varphi[\psi / \sigma]
\end{aligned}$$

Notice that the last step is by $\Gamma \vdash \varphi[\psi/\sigma] \to \psi$.

By the positiveness of φ in σ, we just need to prove that for all $\varphi_1, \dots, \varphi_n$:

$$\Gamma \vdash (\forall \vec{x} \,.\, \alpha_0)[\varphi_1, \dots, \varphi_n] \to \psi[\varphi_1/x_1 \dots \varphi_n/x_n]$$

By (Key-Value) and definition of α_0, the above becomes

$$\Gamma \vdash z_1 \in \varphi_1 \wedge \cdots \wedge z_n \in \varphi_n \wedge \psi[z_1/x_1 \dots z_n/x_n] \to \psi[\varphi_1/x_1 \dots \varphi_n/x_n],$$

which holds by assumption. QED.

What is interesting in the above proof is that we used only (Key-Value) and did not use (Injectivity) and (Product Domain). The last two axioms are used in the proof of Theorem 5.3, where we need to establish an isomorphism between *models* of LFP and MmL. In there, the two axioms are needed to constrain matching μ-logic models.

5.14.2 Proof of Theorem 5.3

We first show that the theory of products $\Gamma^{\mathsf{product}}$ in Definition 5.2 captures precisely the product set $M_s \times M_t$.

Now, we are ready to prove Theorem 5.3.

Proof of Theorem 5.3. The proof is mainly based on the isomorphism between LFP models and matching μ-logic Γ^{LFP}-models. Notice that the (Function) axioms forces symbols in all Γ^{LFP}-models are functions. In addition, we use the axiom $\forall x : \mathsf{Formula}\, \forall y : \mathsf{Formula} \,.\, x = y$ to force that the carrier set of Formula must be a singleton set, say, $\{\star\}$.

(The "if" direction). We follow the same idea as we prove that matching logic captures faithfully FOL (see [2]), we construct from an LFP model $(\{M_s^{\mathsf{LFP}}\}_{s\in S}, \Sigma^{\mathsf{LFP}}, \Pi^{\mathsf{LFP}})$ a corresponding matching μ-logic Γ^{LFP} model, denoted $(\{M_s^{\mathsf{MmL}}\}_{s\in S} \cup \{M_{\mathsf{Formula}}^{\mathsf{MmL}}\}, \Sigma^{\mathsf{MmL}})$ with $M_s^{\mathsf{MmL}} = M_s^{\mathsf{LFP}}$, $M_{\mathsf{Formula}}^{\mathsf{MmL}} = \{\star\}$, and Σ^{MmL} consisting of symbols that are all functions. An LFP valuation ρ^{LFP} derives a corresponding matching μ-logic valuation ρ^{MmL} such that $\rho^{\mathsf{MmL}}(x) = \rho^{\mathsf{LFP}}(x)$ for all LFP (element) variables x and $\rho^{\mathsf{MmL}}(R) = \rho^{\mathsf{LFP}}(R) \times \{\star\}$. Our goal is to prove that for all LFP formulas φ, we have $M^{\mathsf{LFP}}, \rho^{\mathsf{LFP}} \vDash_{\mathsf{LFP}} \varphi$ if and only if $|\varphi|_{M,\rho^{\mathsf{MmL}}} = \{\star\}$.

Firstly, notice that as shown in [2], $|t|_{M,\rho^{\mathsf{MmL}}} = \{\rho^{\mathsf{LFP}}(t)\}$ for all terms t. Therefore, to simplify our notation we uniformly use $\rho(t)$ in LFP and matching μ-logic to mean the evaluation of t. Carry out induction on the structure of φ. The only additional cases (compared with FOL) are $\varphi \equiv R(t_1, \dots, t_n)$ and $\varphi \equiv [\mathsf{lfp}_{R,x_1,\dots,x_n} \psi](t_1, \dots, t_n)$. The first case is easy, as shown in the following reasoning: $M^{\mathsf{LFP}}, \rho^{\mathsf{LFP}} \vDash R(t_1, \dots, t_n)$ iff $(\rho(t_1), \dots, \rho(t_n)) \in \rho^{\mathsf{LFP}}(R)$

iff $(\rho(t_1),\dots,\rho(t_n),\star) \in |R|_{M,\rho^{\mathsf{MmL}}}$ iff $|R(t_1,\dots,t_n)|_{M,\rho^{\mathsf{MmL}}} = \{\star\}$. The second case when $\varphi \equiv [\mathsf{lfp}_{R,x_1,\dots,x_n}\psi](t_1,\dots,t_n)$ is shown as the following reasoning:

$$\begin{aligned}
&M^{\mathsf{LFP}}, \rho^{\mathsf{LFP}} \vDash_{\mathsf{LFP}} [\mathsf{lfp}_{R,x_1,\dots,x_n}\psi](t_1,\dots,t_n)\\
\text{iff } &(\rho(t_1),\dots,\rho(t_n)) \in\\
&\textstyle\bigcap\{\alpha \subseteq M^{\mathsf{LFP}}_{s_1} \times \cdots \times M^{\mathsf{LFP}}_{s_n} \mid \text{for all } a_i \in M^{\mathsf{LFP}}_{s_i}, 1 \le i \le n,\\
&\quad M^{\mathsf{LFP}}, \rho^{\mathsf{LFP}}[\alpha/R, \vec{a}/\vec{x}] \vDash_{\mathsf{LFP}} \psi \text{ implies } (a_1,\dots,a_n) \in \alpha\}\\
\text{iff } &\text{(by induction hypothesis)}\\
&(\rho(t_1),\dots,\rho(t_n)) \in\\
&\textstyle\bigcap\{\alpha \subseteq M^{\mathsf{MmL}}_{s_1} \times \cdots \times M^{\mathsf{MmL}}_{s_n} \mid \text{for all } a_i \in M^{\mathsf{MmL}}_{s_i}, 1 \le i \le n,\\
&\quad |\psi|_{M,(\rho[\alpha/R,\vec{a}/\vec{x}])^{\mathsf{MmL}}} = \{\star\} \text{ implies } (a_1,\dots,a_n) \in \alpha\}\\
\text{iff } &\text{(by definition of } (\rho[\alpha/R,\vec{a}/\vec{x}])^{\mathsf{MmL}})\\
&(\rho(t_1),\dots,\rho(t_n)) \in\\
&\textstyle\bigcap\{\alpha^+ \subseteq M^{\mathsf{MmL}}_{s_1} \times \cdots \times M^{\mathsf{MmL}}_{s_n} \times \{\star\} \mid\\
&\quad \text{for all } a_i \in M^{\mathsf{MmL}}_{s_i}, 1 \le i \le n,\\
&\quad |\psi|_{M,\rho^{\mathsf{MmL}}[\alpha^+/R,\vec{a}/\vec{x}]} = \{\star\} \text{ implies } (a_1,\dots,a_n,\star) \in \alpha^+\}\\
\text{iff } &\text{(by reasoning about sets)}\\
&(\rho(t_1),\dots,\rho(t_n)) \in\\
&\textstyle\bigcap\{\alpha^+ \subseteq M^{\mathsf{MmL}}_{s_1} \times \cdots \times M^{\mathsf{MmL}}_{s_n} \times \{\star\} \mid\\
&\quad \bigcup_{a_i \in M^{\mathsf{MmL}}_{s_i}} (a_1,\dots,a_n, |\psi|_{M,\rho^{\mathsf{MmL}}[\alpha^+/R,\vec{a}/\vec{x}]}) \subseteq \alpha^+\}\\
\text{iff } &\text{(by matching } \mu\text{-logic semantics)}\\
&(\rho(t_1),\dots,\rho(t_n)) \in\\
&|\mu R \colon s_1 \otimes \ldots \otimes s_n \otimes \mathsf{Formula} \,.\, \exists x_1 \ldots \exists x_n \,.\, \langle x_1,\dots,x_n,\psi\rangle|_{M,\rho^{\mathsf{MmL}}},
\end{aligned}$$

and the last statement is equal to $|[\mathsf{lfp}_{R,x_1,\dots,x_n}\psi](t_1,\dots,t_n)|_{M,\rho^{\mathsf{MmL}}}$.

And now we conclude that $\Gamma^{\mathsf{LFP}} \vDash \varphi$ implies $\vDash_{\mathsf{LFP}} \varphi$. Otherwise, there exists an LFP model M^{LFP} and valuation ρ^{LFP} such that $M^{\mathsf{LFP}}, \rho^{\mathsf{LFP}} \nvDash_{\mathsf{LFP}} \varphi$, and this implies that in the Γ^{LFP}-model M^{MmL}, we have $|\varphi|_{M,\rho^{\mathsf{MmL}}} \neq \{\star\}$, meaning that $\Gamma^{\mathsf{LFP}} \nvDash \varphi$.

(The "only if" part). Notice the axiom $\forall x : \mathsf{Formula}\, \forall y : \mathsf{Formula} \,.\, x = y$ forces that $M_{\mathsf{Formula}} = \{\star\}$ must be a singleton set, which ensures that the above translation from an LFP model M^{LFP} to a matching μ-logic model M^{MmL} can go *backward*. Specifically, for every matching

μ-logic function symbol $f \in \Sigma^{\mathsf{MmL}}_{s_1\ldots s_n,s}$, we construct from its interpretation $f_{M^{\mathsf{MmL}}} \colon M_{s_1} \times \cdots \times M_{s_n} \to \mathcal{P}(M_s)$, the corresponding LFP function $f_{M^{\mathsf{LFP}}} \colon M_{s_1} \times \cdots \times M_{s_n} \to M_s$ such that $f_{M^{\mathsf{MmL}}}(a_1, \ldots, a_n) = \{f_{M^{\mathsf{LFP}}}(a_1, \ldots, a_n)\}$. Similarly, for every matching μ-logic function symbol $\pi \in \Sigma^{\mathsf{MmL}}_{s_1\ldots s_n,\mathsf{Formula}}$, we construct from its interpretation $\pi_{M^{\mathsf{MmL}}} \colon M_{s_1} \times \cdots \times M_{s_n} \to \{\emptyset, \{\star\}\}$, the corresponding LFP predicate $\pi_{M^{\mathsf{LFP}}} \subseteq M_{s_1} \times \cdots \times M_{s_n}$, such that $\pi_{M^{\mathsf{LFP}}} \subseteq M_{s_1} \times \cdots \times M_{s_n} = \{(a_1, \ldots, a_n) \mid \pi_{M^{\mathsf{MmL}}}(a_1, \ldots, a_n) = \{\star\}\}$. Then we carry out the same reasoning as in the "if" part. QED.

5.14.3 Proof of Theorem 5.4

Proof. We only need to prove that for every s and h, $s, h \vDash_{\mathsf{SL}} p(x_1, \ldots, x_n)$ iff $h \in |p(x_1, \ldots, x_n)|_{\mathsf{Map},\rho_s}$, where $\rho_s(x) = s(x)$ for all x. We conduct structural induction on φ. The case when $\varphi \equiv p(\varphi_1, \ldots, \varphi_n)$ where p is a recursive predicate is proved directly by the definition of the canonical model Map. The other cases have been proved in [2, Proposition 9.2]. QED.

5.14.4 Proof of Proposition 5.13

Proof of Proposition 5.13. We simply apply definitions. Recall that $s \in \bullet_T(t)$ iff $s \, R \, t$.

(Case "$\bullet$"). $s \in |\bullet\varphi|_{T,\rho}$ iff there exists $t \in |\varphi|_{T,\rho}$ such that $s \in \bullet_T(t)$ iff there exists t such that $s \, R \, t$ and $t \in |\varphi|_{T,\rho}$.

(Case "$\circ$"). $s \in |\circ\varphi|_{T,\rho}$ iff $s \in |\neg\bullet\neg\varphi|_{T,\rho}$ iff $s \notin |\bullet\neg\varphi|_{T,\rho}$ iff (use (Case "$\bullet$")) for all t, $t \in |\neg\varphi|_{T,\rho}$ implies $s \notin \bullet_T(t)$ iff for all t, $s \in \bullet_T(t)$ implies $t \in |\varphi|_{T,\rho}$ iff for all t, $s \, R \, t$ implies $t \in |\varphi|_{T,\rho}$.

(Case "$\diamond$"). Note that $|\diamond\varphi|_{T,\rho} = \bigcap\{A \subseteq S \mid |\varphi \vee \bullet X|_{T,\rho[A/X]} \subseteq A\} = \bigcap\{A \subseteq S \mid |\varphi|_{T,\rho} \cup \bullet_T(A) \subseteq A\}$. On the other hand, $\{s \in S \mid \exists t \in S$ such that $t \in |\varphi|_{T,\rho}$ and $s \, R^* \, t\} = \{s \in S \mid \exists t \in S, \exists n \geq 0$ such that $t \in |\varphi|_{T,\rho}$ and $s \, R^n \, t\} = \{s \in S \mid \exists n \geq 0$ such that $s \in \bullet^n_T(|\varphi|_{T,\rho})\} = \bigcup_{n\geq 0} \bullet^n_T(|\varphi|_{T,\rho})$. Therefore, we just need to prove the two sets:

$$(\eta) \equiv \bigcap\{A \subseteq S \mid |\varphi|_{T,\rho} \cup \bullet_T(A) \subseteq A\}$$
$$(\xi) \equiv \bigcup_{n\geq 0} \bullet^n_T(|\varphi|_{T,\rho})$$

are equal. This can be directly proved by Knaster-Tarski theorem.

(Case "$\Box$"). Similar to (Case "$\diamond$").

(Case "$\varphi_1 \, U \, \varphi_2$"). As in (Case "$\diamond$"), we define two sets:

$$(\eta) \equiv |\varphi_1 \, U \, \varphi_2|_{T,\rho} = \bigcap\{A \subseteq S \mid |\varphi_2|_{T,\rho} \cup (|\varphi_1 \cap \bullet_T(A)|_{T,\rho}) \subseteq A\}$$

$$\begin{aligned}(\xi) \equiv \{s \in S \mid{} & \text{exist } n \geq 0 \text{ and } t_1, \ldots, t_n \in S \text{ such that} \\ & s\, R\, t_1\, R \ldots R\, t_n, \text{ and } s, t_1, \ldots, t_{n-1} \in |\varphi_1|_{T,\rho}, t_n \in |\varphi_2|_{T,\rho}\}\end{aligned}$$

and then use Knaster-Tarski theorem to prove them equal.

(Case "WF"). Again, we define two sets:

$$\begin{aligned}(\eta) &\equiv |\mu X \,.\, \circ X|_{T,\rho} = \bigcap\{A \subseteq S \mid (S \setminus A) \subseteq \bullet_T(S \setminus A)\} \\ (\xi) &\equiv \{s \in S \mid s \text{ has no infintie path}\}\end{aligned}$$

and then use Knaster-Tarski theorem to prove them equal. QED.

5.14.5 Proof of Theorem 5.15

Let us first review some characteristic subclasses of transition systems.

Definition 5.7. A transition system $T = (S, R)$ is:

- *well-founded* if for all $s \in S$, there is no infinite path from s;
- *non-terminating*, if for all $s \in S$ there is $t \in S$ such that $s\, R\, t$.
- *linear*, if for all $s \in S$ and $t_1, t_2 \in S$ such that $s\, R\, t_1$ and $s\, R\, t_2$, then $t_1 = t_2$.

Lemma 5.6. $\vdash_{\mathsf{infLTL}} \varphi$ implies $\Gamma^{\mathsf{infLTL}} \vdash \varphi$.

Proof. We just need to prove that all proof rules in Figure 2.5 can be proved in Γ^{infLTL}.

(Taut) and (MP). Trivial.

($\mathrm{K}_\circ$) and ($\mathrm{N}_\circ$). By Proposition 3.6.

($\mathrm{K}_\Box$) and ($\mathrm{N}_\Box$). Proved by applying (Knaster Tarski) first, followed by simple propositional and modal logic reasoning.

(Fun, "→"). Proved from axiom (Inf) $\bullet\top$ and simple modal logic reasoning.

(Fun, "←"). Exactly axiom (Lin).

(U_1). By (Knaster Tarski) followed by propositional reasoning.

(U_2). By definition of $\varphi_1\, U\, \varphi_2$ as a least fixpoint and (Fun).

(Ind). By (Knaster Tarski). QED.

Lemma 5.7. $s \vDash_{\mathsf{infLTL}} \varphi$ if and only if $s \in |\varphi|_{T,V}$.

Proof. We make two interesting observations. Firstly, it suffices to prove merely the "only if" part. The "if" part follows by considering the "only if" part on $\neg\varphi$.

Secondly, the definition of "$s \vDash_{\mathsf{infLTL}} \varphi$" is an *inductive* one, meaning that "$\vDash_{\mathsf{infLTL}}$" is the least relation that satisfies the five conditions in Definition **??**. To show that "$s \vDash_{\mathsf{infLTL}} \varphi$ implies $s \in |\varphi|_{T,V}$", it suffices to show that $s \in |\varphi|_{T,V}$ satisfies the same conditions. This is easily followed by Proposition 5.13.

Note how interesting that this lemma is proved by applying Knaster-Tarski theorem in the meta-level. QED.

Corollary 5.1. $\Gamma^{\mathsf{infLTL}} \vDash \varphi$ implies $\vDash_{\mathsf{infLTL}} \varphi$.

Proof. Assume the opposite and there exists a transition system $T = (S, R)$ that is linear and non-terminating, a valuation V, and a state $s \in S$ such that $s \nvDash_{\mathsf{infLTL}} \varphi$. By Lemma 5.7, $s \notin |\varphi|_{T,V}$, meaning that $T \nvDash \varphi$. Since T is non-terminating and linear, the axioms (Inf) and (Lin) hold in T, and thus $\Gamma^{\mathsf{infLTL}} \nvDash \varphi$. Contradiction. QED.

Now we are ready to prove Theorem 5.15.

Proof of Theorem 5.15. Use Lemma 5.6 and Corollary 5.1, as well as the soundness of MmL proof system and the completeness of infinite-trace LTL proof system. QED.

5.14.6 Proof of Theorem 5.16

Lemma 5.8. $\vdash_{\mathsf{finLTL}} \varphi$ implies $\Gamma^{\mathsf{finLTL}} \vdash \varphi$.

Proof. We just need to prove all proof rules in Figure 2.6 can be proved by axioms (Fin) and (Lin) in MmL. We skip the ones that have shown in Lemma 5.6.

($\neg\circ$). Proved by axiom (Lin).

(coInd). Use axiom (Fin) $\mu X \,.\, \circ X$ and to prove $\Gamma^{\mathsf{finLTL}} \vdash \mu X \,.\, \circ X \to \varphi$ by (Knaster Tarski).

(Fix). By definition of $\varphi_1 \, W \, \varphi_2$ as a least fixpoint. QED.

Lemma 5.9. $s \vDash_{\mathsf{finLTL}} \varphi$ if and only if $s \in |\varphi|_{T,V}$.

Proof. As in Lemma 5.7, we just need to prove the "only if" part, by showing that $s \in |\varphi|_{T,V}$ satisfies the five conditions in Definition **??**. This is easily followed by Proposition 5.13. The case $\varphi_1 \, W \, \varphi_2$ shall be proved by directly applying matching μ-logic semantics. QED.

Corollary 5.2. $\Gamma^{\mathsf{finLTL}} \vDash \varphi$ implies $\vDash^{\mathsf{finLTL}} \varphi$.

Proof. Assume the opposite and use Lemma 5.9. QED.

Now we can prove Theorem 5.16.

Proof of Theorem 5.16. Use Lemma 5.8 and Corollary 5.2, as well as the soundness of the matching μ-logic proof system and the completeness of finite-trace LTL proof system. QED.

5.14.7 Proof of Theorem 5.17

Lemma 5.10. $\vdash_{\mathsf{CTL}} \varphi$ implies $\Gamma^{\mathsf{CTL}} \vdash \varphi$.

Proof. We just need to prove all CTL rules from the axiom (Inf) in matching μ-logic. We skip the first 7 rules as they are simple. The rest 3 rules can be proved by applying (Knaster Tarski) and use properties in Properties 5.28. QED.

Lemma 5.11. $s \vDash_{\mathsf{CTL}} \varphi$ if and only if $s \in |\varphi|_{T,V}$.

Proof. As in Lemma 5.7, we just need to prove the "only if" part by showing that $s \in |\varphi|_{T,V}$ satisfies all 7 conditions in Definition **??**. The first 5 of them are simple. We show the last two ones about "EU" and "AU".

(Case EU). Assume there exists a path $s_0 s_1 \ldots$ and $k \geq 0$ such that $s_k \in |\varphi_2|_{T,V}$ and $s_0, \ldots, s_{k-1} \in |\varphi_1|_{T,V}$. Our goal is to show $s_0 \in |\varphi_1 \mathsf{EU}\, \varphi_2|_{T,V}$. By semantics of matching μ-logic, $|\varphi_1 \mathsf{EU}\, \varphi_2|_{T,V} = |\mu X \,.\, \varphi_2 \vee (\varphi_1 \wedge \bullet X)|_{T,V} = \bigcap\{A \subseteq S \mid |\varphi_2|_{T,V} \cup (|\varphi_1|_{T,V} \cap \bullet_T(A)) \subseteq A\}$. Therefore, it suffices to prove that $s_0 \in A$ for all $A \subseteq S$ such that $|\varphi_2|_{T,V} \subseteq A$ and $|\varphi_1|_{T,V} \cap \bullet_T(A) \subseteq A$. This is easy, $s_k \in |\varphi_2|_{T,V} \subseteq A$ implies $s_{k-1} \in \bullet_T(s_k)$. Also, $s_{k-1} \in |\varphi_1|_{T,V}$ by assumption. Then $s_{k-1} \in |\varphi_1|_{T,V} \cap \bullet_T(s_k) \subseteq A$. Repeat this procedure for k times and we obtain $s_0 \in A$. Done.

(Case AU). Let us denote $\circ_T(A) = \{s \in S \mid \text{for all } t \in S \text{ such that } s\, R\, t, t \in A\}$ to be the "interpretation" of "all-path next $\circ$" in T. Prove by contradiction. Assume the opposite statement that $s_0 \notin |\varphi_1 \mathsf{AU}\, \varphi_2|_{T,V} = |\mu X \,.\, \varphi_2 \vee (\varphi_1 \wedge \circ X)|_{T,V} = \bigcap\{A \subseteq S \mid |\varphi_2|_{T,V} \cup (|\varphi_1|_{T,V} \cap \circ_T(A)) \subseteq A\}$. This means that there exists $A \subseteq S$ such that $|\varphi_2|_{T,V} \subseteq A$ and $|\varphi_1|_{T,V} \cap \circ_T(A) \subseteq A$, and $s_0 \notin A$. This is going to cause contradiction. Firstly by $|\varphi_2|_{T,V} \subseteq A$, $s_0 \notin |\varphi_2|_{T,V}$, which implies that $s_0 \in |\neg\varphi_2|_{T,V}$. Secondly by $|\varphi_1|_{T,V} \cap \circ_T(A) \subseteq A$, we know that $(S \setminus A) \subseteq |\neg\varphi_1|_{T,V} \cup \bullet_T(S \setminus A)$. Since $s_0 \notin A$, we know either $s_0 \in |\neg\varphi_1|_{T,V}$ or $s_0 \in \bullet_T(S \setminus A)$. If it is the first case, then we have a contradiction as any path starting from s_0 contradicts with the condition. If it is the second case, then there exists a state, say s_1, such that $s_0\, R\, s_1$ and $s_1 \notin A$, which also implies $s_1 \notin |\varphi_2|_{T,V}$. Repeat this process and obtain a sequence of state $s_0 s_1 \ldots$. If the sequence is finite, say $s_0 s_1 \ldots s_n$, then by construction $s_0, \ldots, s_n \notin |\varphi_2|_{T,V}$ and $s_n \in |\neg\varphi_1|_{T,V}$, which is a contradiction to the condition. If the sequence is infinite, then by construction $s_0 s_1 \ldots$ satisfies that $s_0, s_1, \notin |\varphi_2|_{T,V}$, which also contradicts to the condition. QED.

Corollary 5.3. $\Gamma^{\mathsf{CTL}} \vDash \varphi$ implies $\vDash_{\mathsf{CTL}} \varphi$.

Proof. Use Lemma 5.11 and prove by contradiction. Note that it is easy to verify that $T \vDash \Gamma^{\mathsf{CTL}}$ if T is non-terminating. QED.

Now we are ready to prove Theorem 5.17.

Proof of Theorem 5.17. Use Lemma 5.10 and Corollary 5.3, as well as soundness of MmL and completeness of CTL. QED.

5.14.8 Proof of Theorem 5.18

Lemma 5.12. $\vdash_{\mathsf{DL}} \varphi$ implies $\Gamma^{\mathsf{DL}} \vdash \varphi$.

Proof. We just need to prove that all proof rules in **??** can be proved in Γ^{DL}. First of all, rules (DL$_3$) to (DL$_6$) follow from (syntactic sugar) definitions. Rules (Taut) and (MP) are trivial, We only prove (DL$_1$), (DL$_2$), (DL$_7$), and (Gen).

Notice that $[\alpha]\varphi$ is defined a syntactic sugar based on the structure of α. Therefore, we carry out structure induction on α. We should be careful to prevent circular reasoning. Our proving strategy is to prove (Gen) first, and then prove (DL$_1$) and (DL$_2$) simultaneously, and finally prove (DL$_7$).

(Gen). Carry out induction on α. All cases are trivial. Notice the case when $\alpha \equiv \beta^*$ is proved by proving $\Gamma^{\mathsf{DL}} \vdash \varphi \to [\alpha^*]\varphi$ using (Knaster Tarski). After simplification, the goal becomes $\Gamma^{\mathsf{DL}} \vdash \varphi \to [\beta]\varphi$. This is proved by applying induction hypothesis, which shows $\Gamma^{\mathsf{DL}} \vdash [\beta]\varphi$.

(DL$_1$) and (DL$_2$). We prove both rules simultaneously by induction on α. We discuss only interesting cases and skip the trivial ones. (DL$_1$, $\alpha \equiv \beta_1 ; \beta_2$) is proved from induction hypothesis, by applying (Gen) on $[\beta_1]$. (DL$_1$, $\alpha \equiv \beta^*$) is proved by applying (Knaster Tarski), following by applying (DL$_2$, "$\to$") on $[\beta]$. (DL$_2$, $\alpha \equiv \beta^*$, "$\to$") is proved by (Knaster Tarski). (DL$_2$, $\alpha \equiv \beta^*$, "$\leftarrow$") is proved by (Knaster Tarski), followed by (DL$_2$) on $[\beta]$.

(DL$_7$) is proved directly by (Knaster Tarski), followed by (DL$_2$, "$\leftarrow$") on $[\alpha]$.

QED.

We now connect the semantics of DL with the semantics of MmL. First of all, we show that the transition system $T = (S, \{R_a\}_{a \in APgm})$ can be regarded as a Σ^{LTS}-model, where S is the carrier set of State and $APgm$ (the set of atomic programs) is the carrier set of Pgm. The "one-path next $\bullet \in \Sigma_{\mathsf{Pgm}\,\mathsf{State},\mathsf{State}}$ is interpreted *according to DL semantics* for all $t \in S$ and $a \in APgm$:

$$\bullet_T(a, t) = \{s \in S \mid (s, t) \in R_a\}.$$

In addition, valuation $V : AP \to \mathcal{P}(S)$ can be regarded as a matching μ-logic valuation (where we safely ignore the valuations of element variables because they do not appear in DL syntax).

Lemma 5.13. Under the above notations, $[\![\varphi]\!]^T_V = |\varphi|_{T,V}$.

Proof. As in Lemma 5.7, we just need to prove that $[\![\varphi]\!]^T_V \subseteq |\varphi|_{T,V}$ by showing that $|\varphi|_{T,V}$ satisfies the conditions in Definition **??**. The only interesting case is to show $|[\alpha]\varphi|_{T,V} = \{s \in S \mid \text{for all } t \in S, (s,t) \in [\![\alpha]\!]^T_V \text{ implies } t \in |\varphi|_{T,V}\}$. We prove it by carrying out structural induction on the DL program formula α. The case when $\alpha \equiv a$ for $a \in APgm$ is easy. The cases when $\alpha \equiv \beta_1 \,;\, \beta_2$, $\alpha \equiv \beta_1 \cup \beta_2$, and $\alpha \equiv \psi?$ follows directly by basic analysis about sets and using definition of the semantics of DL program formulas. The interesting case is when $\alpha \equiv \beta^*$. In this case we should prove $|[\beta^*]\varphi|_{T,V} = |\nu X \,.\, \varphi \wedge [\beta]X|_{T,V} = \bigcup\{A \mid A \subseteq |\varphi|_{T,V} \cap |[\beta]X|_{T,V[A/X]}\} = \bigcup\{A \mid A \subseteq |\varphi|_{T,V} \cap \{s \mid \text{for all } t, (s,t) \in [\![\beta]\!]^T_V \text{ implies } t \in S\}\} \overset{?}{=} \{s \mid \text{for all } t, (s,t) \in [\![\beta^*]\!]^T_V \text{ implies } t \in |\varphi|_{T,V}\}$ We denote the left-hand side of "$\overset{?}{=}$" as (η) and the right-hand side as (ξ).

To prove $(\eta) = (\xi)$, we prove containment from both directions.

(Case $(\eta) \subseteq (\xi)$). This is proved by considering an $s \in (\eta)$ and show $s \in (\xi)$. By construction of (η), there exists $A \subseteq S$ such that $A \subseteq |\varphi|_{T,V} \cap \{s \mid \text{for all } t, (s,t) \in [\![\beta]\!]^T_V \text{ implies } t \in A\}$, and that $s \in A$. In order to prove $s \in (\xi)$, we assume $t \in S$ such that $(s,t) \in ([\![\beta]\!]^T_V)^*$ and try to prove $t \in |\varphi|_{T,V}$. By definition, there exists $k \geq 0$ and $s_0, \dots, s_k$ such that $s = s_0$, $t = s_k$, and $(s_i, s_{i+1}) \in [\![\beta]\!]^T_V$ for all $0 \leq i < k$. By induction and the property of A, and that $s_0 \in A$, we can prove that $s_0, s_1, \dots, s_k \in |\varphi|_{T,V}$, and thus $t \in |\varphi|_{T,V}$. Done.

(Case $(\xi) \subseteq (\eta)$). Notice that the set η is defined as a greatest fixpoint, so it suffices to show that (ξ) satisfies the condition, i.e., to prove that $(\xi) \subseteq |\varphi|_{T,V} \cap \{s \mid \text{for all } t, (s,t) \in [\![\beta]\!]^T_V \text{ implies } t \in (\xi)\}$. This can be easily proved by the definition of (ξ). Done. QED.

Corollary 5.4. $\Gamma^{\mathsf{DL}} \vDash \varphi$ implies $\vDash_{\mathsf{DL}} \varphi$.

Proof. Use Lemma 5.13, and for the sake of contradiction, assume the opposite. Suppose there exists $T = (S, \{R_a\}_{a \in APgm})$ and a valuation V and a state s such that $s \notin [\![\varphi]\!]^T_V$. We then know $s \notin |\varphi|_{T,V}$, which implies that $T \not\vDash \varphi$. Obviously $T \vDash \Gamma^{\mathsf{DL}}$ as the theory Γ^{DL} contains no addition axioms. This means that $\Gamma^{\mathsf{DL}} \not\vDash \varphi$. QED.

We are ready to prove Theorem 5.18.

Proof of Theorem 5.18. Use Lemma 5.12 and Corollary 5.4, as well as soundness of MmL and completeness of DL. QED.

5.14.9 Proof of Theorem 5.19

As a review, we have defined the following notations:

"one-path next"	$\bullet\varphi$, where $\bullet \in \Sigma_{\mathsf{Cfg},\mathsf{Cfg}}$
"all-path next"	$\circ\varphi \equiv \neg\bullet\neg\varphi$
"eventually"	$\diamond\,\varphi \equiv \mu X \,.\, \varphi \vee \bullet X$
"always"	$\Box\varphi \equiv \nu X \,.\, \varphi \wedge \circ X$
"well-founded"	$\mathsf{WF} \equiv \mu X \,.\, \circ X$
"weak eventually"	$\diamond_w\,\varphi \equiv \nu X \,.\, \varphi \vee \bullet X$

Proposition 5.28. *The following propositions hold:*

1. $\vdash \bullet\bot \leftrightarrow \bot$
2. $\vdash \bullet(\varphi_1 \vee \varphi_2) \leftrightarrow \bullet\varphi_1 \vee \bullet\varphi_2$
3. $\vdash \bullet(\exists x \,.\, \varphi) \leftrightarrow \exists x \,.\, \bullet\varphi$
4. $\vdash \circ\top \leftrightarrow \top$
5. $\vdash \circ(\varphi_1 \wedge \varphi_2) \leftrightarrow \circ\varphi_1 \wedge \circ\varphi_2$
6. $\vdash \circ(\forall x \,.\, \varphi) \leftrightarrow \forall x \,.\, \circ\varphi$
7. $\vdash \varphi \to \diamond\varphi$ *and* $\vdash \bullet\diamond\varphi \to \diamond\varphi$
8. $\vdash \Box\varphi \to \varphi$ *and* $\vdash \Box\varphi \to \circ\Box\varphi$
9. $\vdash \varphi \to \diamond_w\varphi$ *and* $\vdash \bullet\diamond_w\varphi \to \diamond_w\varphi$
10. $\Gamma \vdash \varphi_1 \to \varphi_2$ *implies* $\Gamma \vdash \star\varphi_1 \to \star\varphi_2$ *where* $\star \in \{\bullet, \circ, \diamond, \Box, \diamond_w\}$
11. $\vdash \diamond\bot \leftrightarrow \bot$

12. $\vdash \diamond(\varphi_1 \vee \varphi_2) \leftrightarrow \diamond\varphi_1 \vee \diamond\varphi_2$

13. $\vdash \diamond(\exists x \,.\, \varphi) \leftrightarrow \exists x \,.\, \diamond \varphi$

14. $\vdash \Box\top \leftrightarrow \top$

15. $\vdash \Box(\varphi_1 \wedge \varphi_2) \leftrightarrow \Box\varphi_1 \wedge \Box\varphi_2$

16. $\vdash \Box(\forall x \,.\, \varphi) \leftrightarrow \forall x \,.\, \Box\varphi$

17. $\vdash \Box\varphi \leftrightarrow \neg \diamond \neg\varphi$

18. $\vdash \circ\varphi_1 \wedge \bullet\varphi_2 \to \bullet(\varphi_1 \wedge \varphi_2)$

19. $\vdash \circ(\varphi_1 \to \varphi_2) \wedge \bullet\varphi_1 \to \bullet\varphi_2$

20. $\vdash \diamond_w\varphi \leftrightarrow (\mathsf{WF} \to \diamond\varphi)$

21. $\vdash \diamond_w(\varphi_1 \vee \varphi_2) \leftrightarrow \diamond_w\varphi_1 \vee \diamond_w\varphi_2$

22. $\vdash \diamond_w(\exists x \,.\, \varphi) \leftrightarrow \exists x \,.\, \diamond_w \varphi$

23. $\vdash \star\star\varphi \leftrightarrow \star\varphi$ *where* $\star \in \{\diamond, \Box, \diamond_w\}$

24. $\vdash \mathsf{WF} \leftrightarrow \mu X \,.\, \circ^k X$ *when* $k \geq 1$

25. $\vdash \mathsf{WF} \leftrightarrow \mu X \,.\, \circ\Box X$

26. $\vdash \Box\varphi_1 \wedge \diamond_w\varphi_2 \to \diamond_w(\varphi_1 \wedge \varphi_2)$

27. $\vdash \Box(\varphi_1 \to \varphi_2) \wedge \varphi_1 \to \varphi_2$

Proof. We prove them in order.

(1–3) follows from (Propagation), and (Framing).

(4–6) are proved from (1–3) and that $\circ\varphi \equiv \neg\bullet\neg\varphi$.

(7) is proved by (Pre-Fixpoint) that $\vdash \varphi \vee \bullet \diamond \varphi \to \diamond\varphi$.

(8) is proved by (Pre-Fixpoint) that $\vdash \Box\varphi \to \varphi \wedge \bullet\Box\varphi$.

(9) is proved by (Knaster Tarski) that $\vdash \varphi \vee \bullet \diamond_w \varphi \to \diamond_w\varphi$.

(10, when $\star$ is $\bullet$) is exactly (Framing).

(10, when $\star$ is $\circ$) is exactly Proposition 3.6.

(10, when $\star$ is $\diamond$) requires us to prove $\Gamma \vdash \diamond\varphi_1 \to \diamond\varphi_2$. By (Knaster Tarski), it suffices to prove $\Gamma \vdash \varphi_1 \vee \bullet \diamond \varphi_2 \to \diamond\varphi_2$, which is proved by (7).

(10, when $\star$ is $\Box$) requires us to prove $\Gamma \vdash \Box\varphi_1 \to \Box\varphi_2$. By (Knaster Tarski), it suffices to prove $\Gamma \vdash \Box\varphi_1 \to \varphi_1 \land \bullet\Box\varphi_2$, which is proved by (8).

(10, when $\star$ is $\diamond_w$) requires us to prove $\Gamma \vdash \diamond_w\varphi_1 \to \diamond_w\varphi_2$. By (Knaster Tarski), it suffices to prove $\Gamma \vdash \diamond_w\varphi_1 \to \varphi_1 \lor \bullet \diamond_w \varphi_2$, which is proved by (Pre-Fixpoint).

(11, "$\to$") is proved by (Knaster Tarski).

(11,"$\leftarrow$") is trivial.

(12, "$\to$") is proved by (Knaster Tarski), followed by (2) to propagate "$\bullet$" through "$\lor$", and finished with (7).

(12, "$\leftarrow$") is prove by (10, when $\star$ is $\diamond$).

(13, "$\to$") is proved by (Knaster Tarski), followed by (3) to propagate "$\bullet$" through "$\exists$", and finished with (7).

(13, "$\leftarrow$") is proved by (10, when $\star$ is $\diamond$).

(14–16) are proved similar to (11–13).

(17, both directions) are proved by (Knaster Tarski) followed by (Pre-Fixpoint).

(18) is proved by $\circ\varphi \equiv \neg\bullet\neg\varphi$ and (Propagation).

(19) is proved by (18) followed by (10).

(20, "$\to$") is proved by proving $\vdash \mathsf{WF} \to (\diamond_w\varphi \to \diamond\varphi)$, which is proved by (Knaster Tarski) followed by (19).

(20, "$\leftarrow$") is proved by (Knaster Tarski), followed by (2) to propagate "$\bullet$" through "$\lor$". After some additional propositional reasoning, we obtain two proof goals: $\vdash \diamond\varphi \to \varphi \lor \bullet \diamond \varphi$ and $\vdash \circ\mathsf{WF} \to \mathsf{WF}$. The former is proved by (Knaster Tarski) and the latter is exactly (Pre-Fixpoint).

(21, "$\to$") is proved by applying (20) everywhere followed by (12).

(21, "$\leftarrow$") is proved by (10, when $\star$ is $\diamond_w$).

(22, "$\to$") is proved by applying (20) everywhere followed by (13).

(22, "$\leftarrow$") is proved by (10, when $\star$ is $\diamond_w$).

(23, when $\star$ is $\diamond$, "$\to$") is proved by (Knaster Tarski) followed by (7).

(23, when $\star$ is $\diamond$, "$\leftarrow$") is proved by (7) and (10).

(23, when $\star$ is $\Box$, "$\to$") is proved by (8) and (10).

(23, when $\star$ is $\Box$, "$\leftarrow$") is proved by (Knaster Tarski) followed by (8).

(23, when $\star$ is $\diamond_w$, "$\to$") is proved by applying (Knaster Tarski) first. Then we need to prove $\vdash \diamond_w \diamond_w \varphi \to \varphi \lor \bullet \diamond_w \diamond_w\varphi$. By (Pre-Fixpoint), we know $\vdash \diamond_w \diamond_w \varphi \to \diamond_w\varphi \lor \bullet \diamond_w \diamond_w\varphi$. Thus, it suffices to prove $\vdash \diamond_w\varphi \lor \bullet\diamond_w \diamond_w\varphi \to \varphi \lor \bullet\diamond_w \diamond_w\varphi$. By propositional reasoning, we just need to prove $\vdash \diamond_w\varphi \to \varphi \lor \bullet \diamond_w \diamond_w\varphi$. By (Knaster Tarski), we know $\vdash \diamond_w\varphi \to \varphi \lor \bullet \diamond_w \varphi$, so it suffices to prove $\vdash \varphi \lor \bullet \diamond_w \varphi \to \varphi \lor \bullet \diamond_w \diamond_w\varphi$. Again by propositional reasoning, it suffices to prove $\vdash \bullet \diamond_w \varphi \to \varphi \vee \bullet \diamond_w \diamond_w \varphi$, which can be proved by proving $\vdash \bullet \diamond_w \varphi \to \bullet \diamond_w \diamond_w \varphi$, which is finally proved by (9) and (10).

(23, when $\star$ is $\diamond_w$, "$\leftarrow$") is proved by (9) and (10).

Note it is sufficient to prove (24) only for the case $k = 1$.

(24, "$\to$") is proved by applying (Knaster Tarski) and (Pre-Fixpoint) first. Then we need to prove $\vdash \mu X \,.\, \circ\circ X \to \circ \mu X \,.\, \circ\circ X$. Apply (Knaster Tarski) again, and finished by (Pre-Fixpoint).

(24, "$\leftarrow$") is proved by applying (Knaster Tarski) followed by (Pre-Fixpoint).

(25, "$\to$") is proved by applying (Knaster Tarski) followed by (Pre-Fixpoint). Then we obtain $\vdash \mu X \,.\, \circ\Box X \to \Box \mu X \,.\, \circ\Box X$. Apply (Knaster Tarski) on $\Box$, and we obtain $\vdash \mu X \,.\, \circ\Box X \to \circ\Box \mu X \,.\, \circ\Box X$, finished by (Pre-Fixpoint).

(25, "$\leftarrow$") is proved by (8), (10), and then apply Lemma 4.4.

(26) is proved by applying (Knaster Tarski) firstly. After propositional reasoning, we obtain two goals: $\vdash \Box\varphi_1 \wedge \diamond_w \varphi_2 \to \varphi_1 \vee \bullet(\Box\varphi_1 \wedge \diamond_w \varphi_2)$ and $\vdash \Box\varphi_1 \wedge \diamond_w \varphi_2 \to \varphi_2 \vee \bullet(\Box\varphi_1 \wedge \diamond_w \varphi_2)$. The first goal is easily proved by (8). The second goal is by unfolding "$\diamond_w \varphi_2$" and "$\Box\varphi_1$", and then use (18).

(27) is proved by (8). QED.

Lemma 5.14. $A \vdash_C \varphi_1 \Rightarrow \varphi_2$ implies $\Gamma^{\mathsf{RL}} \vdash \mathrm{RL2MmL}(A \vdash_C \varphi_1 \Rightarrow \varphi_2)$.

Proof. We need to prove that all reachability logic proof rules in Figure 2.12 are provable in matching μ-logic.

(Axiom). We prove for the case when $C \neq \emptyset$. The case when $C = \emptyset$ is the same. Our goal, after translation, is $\Gamma^{\mathsf{RL}} \vdash \forall \boxdot A \wedge \forall \boxdot C \to (\varphi_1 \to \bullet \diamond_w \varphi_2)$. By assumption, $\varphi_1 \Rightarrow \varphi_2 \in A$, and thus we just need to prove $\Gamma^{\mathsf{RL}} \vdash \forall(\varphi_1 \to \bullet \diamond_w \varphi_2) \to (\varphi_1 \to \bullet \diamond_w \varphi_2)$, which is trivial by FOL reasoning.

(Reflexivity). Notice that $C = \emptyset$ in this rule. Our goal, after translation, is $\Gamma^{\mathsf{RL}} \vdash \forall \boxdot A \to (\varphi \to \diamond_w \varphi)$, which is true by Proposition 5.28.

(Transitivity, $C = \emptyset$). Our goal, after translation, is $\Gamma^{\mathsf{RL}} \vdash \forall \boxdot A \to (\varphi_1 \to \diamond_w \varphi_3)$. Our two assumptions are $\Gamma^{\mathsf{RL}} \vdash \forall \boxdot A \to (\varphi_1 \to \diamond_w \varphi_2)$ and $\Gamma^{\mathsf{RL}} \vdash \forall \boxdot A \to (\varphi_2 \to \diamond_w \varphi_3)$. From the latter assumption and Proposition 5.28, we have $\Gamma^{\mathsf{RL}} \vdash \forall \boxdot A \to (\diamond_w \varphi_2 \to \diamond_w \diamond_w \varphi_3)$, and then by propositional reasoning and the former assumption we have $\Gamma^{\mathsf{RL}} \vdash \forall \boxdot A \to (\varphi_1 \to \diamond_w \diamond_w \varphi_3)$. Finally, by Proposition 5.28 we have $\Gamma^{\mathsf{RL}} \vdash \forall \boxdot A \to (\varphi_1 \to \diamond_w \varphi_3)$, which is what we want to prove.

(Transitivity, $C \neq \emptyset$). Our goal, after translation, is $\Gamma^{\mathsf{RL}} \vdash \forall \boxdot A \wedge \forall \circ \boxdot C \to (\varphi_1 \to \bullet \diamond_w \varphi_3)$. Our two assumptions are $\Gamma^{\mathsf{RL}} \vdash \forall \boxdot A \wedge \forall \circ \boxdot C \to (\varphi_1 \to \bullet \diamond_w \varphi_2)$ and $\Gamma^{\mathsf{RL}} \vdash \forall \boxdot A \wedge \forall \boxdot C \to (\varphi_2 \to \diamond_w \varphi_3)$. From the first assumption, we have $\Gamma^{\mathsf{RL}} \vdash \forall \boxdot A \wedge \forall \circ \boxdot C \wedge \varphi_1 \to$

$\forall\boxdot A \wedge \forall\circ\boxdot C \wedge \bullet \diamond_w \varphi_2$, and thus by propositional reasoning, it suffices to prove that $\Gamma^{\mathsf{RL}} \vdash \forall\boxdot A \wedge \forall\circ\boxdot C \wedge \bullet\diamond_w \varphi_2 \to \bullet\diamond_w \varphi_3$. From the second assumption and Proposition 5.28(10), we know that $\Gamma^{\mathsf{RL}} \vdash \bullet \diamond_w (\forall\boxdot A \wedge \forall\boxdot C \wedge \varphi_2) \to \bullet \diamond_w \diamond_w \varphi_3$, which by Proposition 5.28(23), implies $\Gamma^{\mathsf{RL}} \vdash \bullet \diamond_w (\forall\boxdot A \wedge \forall\boxdot C \wedge \varphi_2) \to \bullet \diamond_w \varphi_3$. Then, it suffices to prove $\Gamma^{\mathsf{RL}} \vdash \forall\boxdot A \wedge \forall\circ\boxdot C \wedge \bullet \diamond_w \varphi_2 \to \bullet \diamond_w (\forall\boxdot A \wedge \forall\boxdot C \wedge \varphi_2)$. The rest is easy, since by Proposition 5.28(8), we just need to prove $\Gamma^{\mathsf{RL}} \vdash \forall\circ\boxdot A \wedge \forall\circ\boxdot C \wedge \bullet \diamond_w \varphi_2 \to \bullet \diamond_w (\forall\boxdot A \wedge \forall\boxdot C \wedge \varphi_2)$, which then by Proposition 5.28(18) becomes $\Gamma^{\mathsf{RL}} \vdash \bullet(\forall\boxdot A \wedge \forall\boxdot C \wedge \diamond_w \varphi_2) \to \bullet \diamond_w (\forall\boxdot A \wedge \forall\boxdot C \wedge \varphi_2)$, and then by Proposition 5.28(10) becomes $\Gamma^{\mathsf{RL}} \vdash \forall\boxdot A \wedge \forall\boxdot C \wedge \diamond_w \varphi_2 \to \diamond_w(\forall\boxdot A \wedge \forall\boxdot C \wedge \varphi_2)$, which is proved by Proposition 5.28(26).

(Logic Framing). We prove for the case when $C \neq \emptyset$. The case when $C = \emptyset$ is the same. Our goal, after translation, is $\Gamma^{\mathsf{RL}} \vdash \forall\boxdot A \wedge \forall\circ\boxdot C \to (\varphi_1 \wedge \psi \to \bullet \diamond_w (\varphi_2 \wedge \psi))$. Our assumption is $\Gamma^{\mathsf{RL}} \vdash \forall\boxdot A \wedge \forall\circ\boxdot C \to (\varphi_1 \to \bullet \diamond_w \varphi_2)$. Notice that FOL formula ψ is a predicate pattern, so $\vdash \bullet \diamond_w (\varphi_2 \wedge \psi) \leftrightarrow (\bullet \diamond_w \varphi_2) \wedge \psi$, and the rest is by propositional reasoning. The condition that ψ is a FOL formula (and thus a predicate pattern) is crucial to propagate ψ throughout its context.

(Consequence). This is the only rule where axioms in Γ^{RL} is actually used. Again, we prove for the case $C \neq \emptyset$ as the case when $C = \emptyset$ is the same. Our goal, after translation, is $\Gamma^{\mathsf{RL}} \vdash \forall\boxdot A \wedge \forall\circ\boxdot C \to (\varphi_1 \to \bullet \diamond_w \varphi_2)$. Our three assumptions include $M^{\mathsf{Cfg}} \vDash \varphi_1 \to \varphi_1'$, $M^{\mathsf{Cfg}} \vDash \varphi_2' \to \varphi_2$, and $\Gamma^{\mathsf{RL}} \vdash \forall\boxdot A \wedge \forall\circ\boxdot C \to (\varphi_1' \to \bullet \diamond_w \varphi_2')$. Notice that by definition of Γ^{RL}, we know immediately that $\varphi_1 \to \varphi_1' \in \Gamma^{\mathsf{RL}}$ and $\varphi_2' \to \varphi_2 \in \Gamma^{\mathsf{RL}}$. The rest of the proof is simply by Proposition 5.28(10) and some propositional reasoning.

(Case Analysis). Simply by some propositional reasoning.

(Abstraction). Simply by some FOL reasoning. Notice that $\forall\boxdot A$ and $\forall\boxdot C$ are closed patterns.

(Circularity). We prove for the case when $C \neq \emptyset$, as the case when $C = \emptyset$ is the same. Our goal, after translation, is $\Gamma^{\mathsf{RL}} \vdash \forall\boxdot A \wedge \forall\circ\boxdot C \to (\varphi_1 \to \bullet \diamond_w \varphi_2)$. By FOL reasoning and Proposition 5.28(20,2,25), the goal becomes $\Gamma^{\mathsf{RL}} \vdash \mu X \,.\, \circ\Box X \to \forall\boxdot A \wedge \forall\circ\boxdot C \to \forall(\varphi_1 \to \bullet \diamond_w \varphi_2)$. By (Knaster Tarski) and some FOL reasoning, it suffices to prove $\Gamma^{\mathsf{RL}} \vdash \circ\Box(\forall\boxdot A \wedge \forall\circ\boxdot C \to \forall(\varphi_1 \to \bullet \diamond_w \varphi_2)) \wedge \forall\boxdot A \wedge \forall\circ\boxdot C \to (\varphi_1 \to \bullet \diamond_w \varphi_2)$. Our assumption, after translation, is $\Gamma^{\mathsf{RL}} \vdash \forall\boxdot A \wedge \forall\circ\boxdot C \wedge \forall\circ(\varphi_1 \to \bullet\diamond_w \varphi_2) \to (\varphi_1 \to \bullet\diamond_w \varphi_2)$, so it suffices to prove $\Gamma^{\mathsf{RL}} \circ\Box(\forall\boxdot A \wedge \forall\circ\boxdot C \to \forall(\varphi_1 \to \bullet\diamond_w \varphi_2)) \wedge \forall\boxdot A \wedge \forall\circ\boxdot C \to \forall\boxdot A \wedge \forall\circ\boxdot C \wedge \forall\circ(\varphi_1 \to \bullet\diamond_w \varphi_2)$, which by some propositional reasoning becomes $\Gamma^{\mathsf{RL}} \vdash \circ\Box(\forall\boxdot A \wedge \forall\circ\boxdot C \to \forall(\varphi_1 \to \bullet \diamond_w \varphi_2)) \wedge \forall\boxdot A \wedge \forall\circ\boxdot C \to \forall\circ(\varphi_1 \to \bullet \diamond_w \varphi_2)$. By Proposition 5.28(8), it becomes $\Gamma^{\mathsf{RL}} \vdash \circ\Box(\forall\boxdot A \wedge \forall\circ\boxdot C \to \forall(\varphi_1 \to \bullet \diamond_w \varphi_2)) \wedge \circ\forall\boxdot A \wedge \circ\forall\circ\boxdot C \to \forall\circ(\varphi_1 \to \bullet \diamond_w \varphi_2)$, and by Proposition 5.28(5,6,10), it becomes $\Gamma^{\mathsf{RL}} \vdash \Box(\forall\boxdot A \wedge \forall\circ\boxdot C \to \forall(\varphi_1 \to \bullet \diamond_w \varphi_2)) \wedge \forall\boxdot A \wedge \forall\circ\boxdot C \to \forall(\varphi_1 \to \bullet \diamond_w \varphi_2)$, which is proved by Proposition 5.28(27). QED.

Corollary 5.5. $S \vdash_{\emptyset} \varphi_1 \Rightarrow \varphi_2$ implies $\Gamma^{\mathsf{RL}} \vdash \mathrm{RL2MmL}(S \vdash_{\emptyset} \varphi_1 \Rightarrow \varphi_2)$.

Proof. Let $A = S$ and $C = \emptyset$ in Lemma 5.14. QED.

Lemma 5.15. $\Gamma^{\mathsf{RL}} \vDash \mathrm{RL2MmL}(S \vdash_{\emptyset} \varphi_1 \Rightarrow \varphi_2)$ implies $S \vDash_{\mathsf{RL}} \varphi_1 \Rightarrow \varphi_2$.

Proof. Let $T = (M^{\mathsf{Cfg}}_{\mathsf{Cfg}}, R)$ be the transition system that is yielded by S. We tactically use the same letter T to mean the extended Σ^{RL}-model M^{Cfg} with $\bullet \in \Sigma_{\mathsf{Cfg},\mathsf{Cfg}}$ be interested as the transition relation R. Then $T \vDash \Gamma^{\mathsf{RL}}$, because all axioms in Γ^{RL} are about only the configuration model M^{Cfg} and says nothing about the transition relation R. Since $M^{\mathsf{Cfg}} \vDash \Gamma^{\mathsf{Cfg}}$ (by definition), then $T \vDash \Gamma^{\mathsf{Cfg}}$. By condition of the lemma, $T \vDash \mathrm{RL2MmL}(S \vdash_{\emptyset} \varphi_1 \Rightarrow \varphi_2)$, i.e., $T \vDash \forall \boxdot S \to \varphi_1 \to \diamond_w \varphi_2$. By construction of T, for all rules $\psi_1 \Rightarrow \psi_2 \in S$, we have $T \vDash \psi_1 \to \bullet \psi_2$ (in MmL), which implies $T \vDash \forall \Box (\psi_1 \to \diamond_w \psi_2)$, meaning that $T \vDash \forall \boxdot S$. Therefore, $T \vDash \varphi_1 \to \diamond_w \varphi_2$ (in MmL), which is exactly the same meaning as $T \vDash_{\mathsf{RL}} \varphi_1 \Rightarrow \varphi_2$ (in RL). QED.

Finally, we are ready to prove Theorem 5.19.

Proof of Theorem 5.19. Following the same roadmap as in the proof of Theorem 5.14, where (2) ⇒ (3) is given by Corollary 5.5 and (5) ⇒ (6) is given by Lemma 5.15. The rest is by the sound and (relative) completeness of RL. Notice that technical assumptions of [11] are needed for the completeness result of RL. QED.

Chapter 6: REASONING ABOUT FIXPOINTS IN MATCHING μ-LOGIC

Automation of fixpoint reasoning has been extensively studied for various mathematical structures, logical formalisms, and computational domains, resulting in specialized fixpoint provers and proof techniques for heaps [23, 88, 89, 90, 91, 92], for streams [93], for term algebras [42], for temporal properties [94], for program reachability correctness [11], and for many other systems and inductive/coinductive properties. However, in spite of great theoretical and practical interest, there is no unifying framework for automated fixpoint reasoning.

Using matching μ-logic, we envision a unifying automated proof framework for fixpoint reasoning, as shown in Figure 6.1. Proofs are done using a fixed set of proof rules that accomplish fixpoint reasoning, in addition to standard FOL reasoning, domain reasoning, frame reasoning, context reasoning, etc, for matching μ-logic, independently of the underlying theory. This way, automated reasoning becomes proof search over the fixed set of proof rules, taking as input a logical theory Γ^L that defines/encodes a certain logical system or programming language L in matching μ-logic. For efficiency, the framework implements various proof strategies as heuristics that guide the proof search, each strategy optimizing formal reasoning within a subset of logical theories.

We will present a prototype implementation of a unified proof framework for automating fixpoint reasoning based on matching μ-logic. We have seen in Chapter 5 that matching μ-logic has good expressive power and can serve as a foundation for a variety of logical systems, including LFP, modal logic, modal μ-calculus, temporal logics (LTL, CTL, etc.), and separation logic. In addition, matching μ-logic patterns admit compact syntax and convenient notations, which allow us to encode formulas in other logical systems almost verbatim.

Our unifying proof framework consists of three main reasoning modules: fixpoint, frame, and context (also illustrated in Figure 6.1). The fixpoint reasoning module is the main one; the other two are to help fixpoint reasoning work properly. Note that these three modules are generic, that is, they work with all theories. Therefore, they accomplish fixpoint reasoning, frame reasoning, and context reasoning for *all* logical systems defined as theories in matching μ-logic.

The main challenge behind developing such a unifying proof framework is that the Hilbert-style proof system $\mathcal{H}_\mu$ of matching μ-logic in Figure 4.1 is too fine-grained to be amenable for automation. For example, consider (Modus Ponens), which says that "$\vdash \varphi \to \psi$ and $\vdash \varphi$ implies $\vdash \psi$". (Modus Ponens) requires the prover to guess a premise φ, which does not

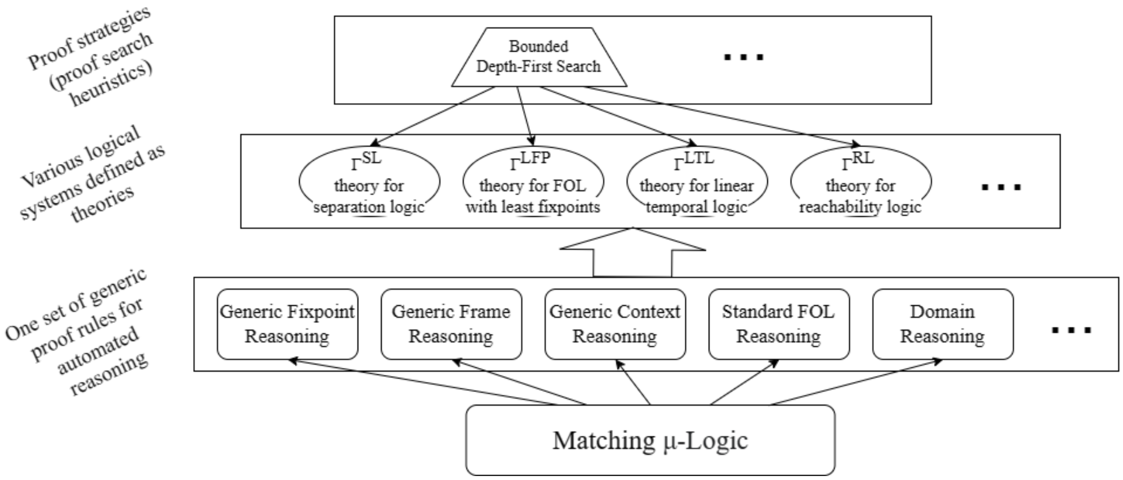

Proof strategies (proof search heuristics)
Various logical systems defined as theories
One set of generic proof rules for automated reasoning
Bounded Depth-First Search
...
Γ^SL theory for separation logic
Γ^LFP theory for FOL with least fixpoints
Γ^LTL theory for linear temporal logic
Γ^RL theory for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

Figure 6.1: Unifying Proof Framework Based on Matching μ-Logic

bode well with automation. When it comes to fixpoint reasoning, the (Knaster Tarski) proof rule (also called Park induction [95]) for fixpoint reasoning

$$\text{(Knaster Tarski)} \qquad \frac{\varphi[\psi/X] \to \psi}{(\mu X \,.\, \varphi) \to \psi}$$

is limited to handling the cases where the left-hand side of the proof goal is a standalone least fixpoint. It cannot be directly applied to proof goals in LFP or SL, such as $\mathit{ll}(x,y) * \mathit{list}(y) \to \mathit{list}(x)$ (see Section 6.1.2), where the left-hand side $C[\mathit{ll}(x,y)]$ contains the fixpoint $\mathit{ll}(x,y)$ within a context $C[\square] \equiv \square * \mathit{list}(y)$. An indirect application is possible *in theory*, but it involves sophisticated, ad-hoc reasoning to eliminate the context C from the left-hand side, which cannot be efficiently automated.

Our fixpoint module addresses the above challenge by proposing a context-driven fixpoint proof rule, (kt), shown in Figure 6.2. (kt) is a sequential composition of several proof rules that first (wrap) context C within the right-hand side ψ, written $C \multimap \psi$, and eliminate it from the left-hand side, then apply inductive reasoning, and finally (unwrap) C and restore it on the left-hand side. The pattern $C \multimap \psi$, called *contextual implication*, is expressible in matching μ-logic and intuitively defines all the elements which in context C satisfy ψ. The fixpoint module therefore makes contexts explicitly occur as conditions in proof goals. Sometimes the context conditions are needed to discharge a proof goal, other times not. The frame and context reasoning modules help to eliminate contexts from proof goals. Specifically, frame reasoning is used when the context is unnecessary: it reduces $\vdash C[\varphi] \to C[\psi]$ to $\vdash \varphi \to \psi$. On the other hand, context reasoning is used when the context is needed in order to discharge the proof goal, by allowing us to derive $\vdash C[C \multimap \psi] \to \psi$. We shall discuss and analyze the frame and context reasoning in detail in Section 6.1.

We have not implemented any smart proof strategies or proof search heuristics, but only a naive bounded depth-first search (DFS) algorithm. Our evaluation on the SL-COMP benchmark shows that the naive bounded-DFS strategy can prove 90% of the properties without frame reasoning, and 95% with frame reasoning (Section 6.4). This was surprising, because it would place our generic proof framework in the third place in the SL-COMP competition, among dozens of specialized provers developed specifically for SL and heap reasoning. However, the remaining 5% properties appear to require complex, SL-specific reasoning, which is clearly beyond the ability of our generic framework. We have also considered only a small number of LFP and LTL proofs, which could all be done using the same simplistic bounded-DFS strategy; more powerful proof strategies will certainly be needed for more complex proofs and will be developed as part of future work.

To evaluate our unified proof framework and prototype implementation, we consider four representative logical systems for fixpoint reasoning: FOL with least fixpoints (LFP), separation logic (SL), linear temporal logic (LTL), and reachability logic (RL), all of which have been introduced in Chapters 2 and 5. We pick these four logics for their representativeness. LFP is the canonical logic for fixpoint reasoning in the first-order domain. SL is the representative logic for reasoning about data-manipulating programs with pointers. LTL is the temporal logic of choice for model checkers of infinite-trace systems, e.g., SPIN [94]. RL is a language-parametric generalization of Hoare logic where the programming language semantics is given as an input theory and partial correctness is specified and proved as a reachability rule $\varphi_{pre} \Rightarrow \psi_{post}$. These four logics therefore represent relevant instances of fixpoint reasoning across different and important domains. We believe that they form a good benchmark for evaluating a unified proof framework for fixpoint reasoning, so we set ourselves the long-term goal to support *all* of them. We will give special emphases to SL in this this, however, because it gathered much attention in recent years that resulted in several automated SL provers and its own international competition SL-COMP [96].

It would be unreasonable to hope at such an incipient stage that a generic automated prover can be superior to the state-of-the-art domain-specific provers and algorithmic decision procedures for all four logics, on all existing challenging benachmarks in their respective domains. Therefore, for each of the domains, we set ourselves a limited objective. For SL, the goal was to prove all the 280 benchmark properties collected by SL-COMP in the problem set `qf_shid_entl` dedicated to inductive reasoning. For LTL, the goal was to prove the axioms about the modal operators "always" $\Box\varphi$ and "until" $\varphi_1 \, U \, \varphi_2$ in its complete proof system. For LFP and RL, our goal was to verify a simple imperative program `sum` that computes the total of 1 to input n using both the LFP and RL encodings, and show that it returns the correct sum $n(n+1)/2$ on termination. We report what we have done in pushing towards the above goals, and discuss the difficulties that we met, and the lessons we learned.

6.1 AUTOMATED PROOF FRAMEWORK FOR MATCHING μ-LOGIC

We will propose a new set of higher-level proof rules (as shown in Figure 6.2) that aim at proof automation. The generic matching μ-logic prover simply runs a simple bounded DFS algorithm over the higher-level proof rules. We first give an overview of the three key reasoning modules offered by the automated proof rules in Section 6.1.1 and then explain all proof rules in detail in Section 6.1.4.

6.1.1 Fixpoint reasoning module

As discussed above, the existing (Knaster Tarski) rule has several limitations due to its general nature, making it impractical for automation. Therefore, we consider two specialized proof rules, (lfp) and (gfp), explained below. Let P be a recursive symbol defined by

$$P(\tilde{x}) =_{\mathbf{lfp}} \exists \tilde{x}_1 \,.\, \varphi_1(\tilde{x}, \tilde{x}_1) \vee \cdots \vee \exists \tilde{x}_m \,.\, \varphi_m(\tilde{x}, \tilde{x}_m)$$

where $\tilde{x}, \tilde{x}_1, \ldots, \tilde{x}_m$ are variable vectors. To prove $\vdash P(\tilde{x}) \to \psi$ for some property ψ, the proof rule (lfp) firstly unfolds $P(\tilde{x})$ according to its definition, and secondly replaces each recursive occurrence $P(\tilde{y})$ (whose arguments $\tilde{y}$ might be different from the original arguments $\tilde{x}$) in φ_i by $\psi[\tilde{y}/\tilde{x}]$, i.e., the result of substituting in ψ the new arguments $\tilde{y}$ for the original arguments $\tilde{x}$. Let us denote the result of substituting each φ_i as $\varphi_i[\psi/P]$. In summary, (lfp) is the following rule (also shown in Figure 6.2):

$$(\textsc{lfp}) \qquad \frac{\exists \tilde{x}_1 \,.\, \varphi_1[\psi/P] \to \psi \quad \cdots \quad \exists \tilde{x}_m \,.\, \varphi_m[\psi/P] \to \psi}{P(\tilde{x}) \to \psi} \tag{6.1}$$

Note that (lfp) generates m new sub-goals (above the bar), each corresponding to one case in the definition of P. All sub-goals have the same, original property ψ on the right-hand side. Intuitively, (lfp) is a logical incarnation of the induction principle that consists of case analysis (according to the definition of p) and inductive hypotheses (i.e., replacing p by the intended property ψ on the left-hand side).

6.1.2 Context reasoning module and contextual implication

Although (LFP) is more syntax-driven than the original (KNASTER TARSKI) rule, it still has limitations. We illustrate them using a simple example in SL

$$\vdash \mathit{ll}(x,y) * \mathit{list}(y) \to \mathit{list}(x) \tag{6.2}$$

where ll and list are defined as recursive predicates:

$$\begin{aligned} \mathit{ll}(x,y) &=_{\mathbf{lfp}} \mathsf{emp} \wedge x = y \vee \exists z \,.\, x \mapsto z * \mathit{ll}(z,y) \\ \mathit{list}(x) &=_{\mathbf{lfp}} \mathsf{emp} \wedge x = 0 \vee \exists z \,.\, x \mapsto z * \mathit{list}(z) \end{aligned}$$

Intuitively, $\mathit{ll}(x,y)$ states that there is a singly-linked list from x to y and $\mathit{list}(x)$ equals to $\mathit{ll}(x,0)$. Clearly, (LFP) cannot be applied directly to (6.2), because the left-hand side is not a recursive symbol, but a larger pattern $\mathit{ll}(x,y) * \mathit{list}(y)$ in which the recursive pattern $\mathit{ll}(x,y)$ occurs. In other words, $\mathit{ll}(x,y)$ occurs within a context in the left-hand side. Let $C[\square] \equiv \square * \mathit{list}(y)$ be the context pattern where $\square$ is a distinguished hole variable. We rewrite proof goal (6.2) to the following form using context C:

$$\vdash C[\mathit{ll}(x,y)] \to \mathit{list}(x) \tag{6.3}$$

Introducing contexts allows us to examine the limitations of rule (LFP) from a more structural point of view. Clearly, (LFP) can only be applied when C is the identity context, i.e., $C_{\mathit{id}}[\square] \equiv \square$, but as we have seen above, in practice recursive patterns often occur within a non-identity context, so a major challenge in applying (LFP) in automated fixpoint reasoning is to handle such non-identity contexts in a systematic way.

To solve the above challenge, we propose an important concept called contextual implication. Recall that a context C is a pattern with a distinguished variable denoted h, called the hole variable. Note that we do not use the standard notation $\square$ to denote the hole variable, to not confuse it with the "always" operator $\square\varphi$ in LTL. We write $C[\varphi]$ as the substitution $C[\varphi/h]$. We say that C is a structure context if $C \equiv t \wedge \psi$ where t is an application context (Definition 3.1) and ψ is a predicate (i.e., patterns equivalent to $\top$ or $\bot$). For example, $h * \mathit{list}(y) \wedge y > 1$ is a structure context (w.r.t. h) because separating conjunction $*$ is a symbol in matching μ-logic (see Section 5.3). All contexts discussed here are structure contexts.

A structure context C is extensive in the hole position, in the following sense. An element a matches $C[\varphi]$ where C is a structure pattern and φ is any pattern plugged into the hole, if and only if there exists an element a_0 that matches φ such that a equals $C[a_0]$. In other words, matching the entire structure $C[\varphi]$ can be reduced to matching the local structure φ and the local reasoning we make about φ at the hole position can be lifted to the entire structure $C[\varphi]$. Therefore, structure contexts allows us to do contextual reasoning.

Let $C[\square]$ be a structure context and ψ be a pattern. We define contextual implication w.r.t. C and ψ as the pattern whose matching elements satisfy ψ if plugged into C.

Definition 6.1. We define *contextual implication* $C \multimap \psi \equiv \exists h \,.\, h \wedge (C[h] \subseteq \psi)$.

Recall that in matching μ-logic, $\exists$ means set union. Thus, $C \multimap \psi$ is the pattern matched by all h such that $C[h] \subseteq \psi$ holds, i.e., when plugged in C, the result $C[h]$ satisfies property ψ. The following is a useful result about contextual implications for structure contexts C:

$$\vdash C[\varphi] \to \psi \quad \underset{\text{(UNWRAP) context } C}{\overset{\text{(WRAP) context } C}{\rightleftharpoons}} \quad \vdash \varphi \to (C \multimap \psi)$$

Note that $C \multimap \psi$ is a regular pattern defined using the syntax of matching μ-logic. It is not an extension of matching μ-logic, but simply a convenient use of the existing expressiveness of patterns to simplify (and automate) formal reasoning by "pulling the target out of its context".

Now, we revisit the SL example and look at proof goal (6.3). By wrapping the structure context $C[\square] = h * \mathit{list}(y)$, we transform it to the following equivalent goal, to which (LFP) can be applied:

$$\vdash \mathit{ll}(x, y) \to (C \multimap \mathit{list}(x)) \quad \text{where } C[\square] = h * \mathit{list}(y)$$

This way, contextual implication helps address the limitations of (LFP) by offering a systematic and general method to wrap/unwrap any contexts, making proof automation based on (LFP) possible.

We conclude the discussion on contextual implication with two remarks. Firstly, after context C is wrapped, the right-hand side becomes $C \multimap \psi$, which by (LFP) will be moved back to left-hand side and replace the recursive occurrences of the recursive pattern (see Equation (6.1), where ψ becomes $C \multimap \psi$). Therefore, we need a set of proof rules to handle and match those contextual implications that occur on the left-hand side using pattern matching. This is explained in detail in Section 6.1.4.

The second remark is that our contextual implication generalizes separating implication $\varphi \mathrel{-\!\!*} \psi$ (the "magic wand") in SL. Indeed, let context $C_\varphi[\square] = \square * \varphi$, then we have $\varphi \mathrel{-\!\!*} \psi = C_\varphi \multimap \psi$. In other words, SL magic wand is a special instance of contextual implication, where the underlying theory is Γ^{SL} and context $C[\square]$ has the specific form $\square * \varphi$ where h occurs immediately below the top-level $*$ operator, and the SL proof rule (ADJ) [22, pp. 5], $\vdash \varphi_1 * \varphi \to \psi$ iff $\vdash \varphi_1 \to (\varphi \mathrel{-\!\!*} \psi)$, is also a special instance of (WRAP) and (UNWRAP). However, contextual implications are more general, because they can be applied to any ML theories and any complex contexts $C[\square]$, e.g., to entire program configurations (see Section 2.14) not only heaps.

6.1.3 Frame reasoning module

Another advantage of having an explicit notion of context as shown above, is that frame reasoning can be generalized to all structure contexts C. In the following, we compare the frame reasoning in separation logic for heap contexts (left, also called (MONOTONE) in [22]) and the general frame reasoning in matching μ-logic for any contexts C (right):

$$\text{(FRAME) rule in SL} \quad \frac{\varphi \to \psi}{\varphi * \varphi_{rest} \to \psi * \varphi_{rest}} \qquad \text{Our (FRAME) rule} \quad \frac{\varphi \to \psi}{C[\varphi] \to C[\psi]}$$

Clearly, the SL (FRAME) rule a special instance of our (FRAME) rule, where $C[\square] \equiv \square * \varphi_{rest}$. Our (FRAME) rule is more general and can be applied to any theories and complex contexts.

We conclude the discussion on frame reasoning with a remark about framing for Hoare-style program correctness using SL as an assertion logic, which has the following form:

$$\text{(FRAME ON PROGRAMS)} \quad \frac{\varphi\ \{\texttt{code}\}\ \psi}{\varphi * \varphi_{rest}\ \{\texttt{code}\}\ \psi * \varphi_{rest}} \quad \text{if } \texttt{code} \text{ does not modify } V_{rest}$$

where $V_{rest} = \mathit{freeVar}(\varphi_{rest})$. If we instantiate `code` by the idle program `skip`, then (FRAME) in SL becomes an instance of (FRAME ON PROGRAMS). While (FRAME ON PROGRAMS) is certainly convenient in practice, we would like to point out that it is language-specific and generally unsound. Indeed, the rule and its side condition itself suggest that the language has a heap and `code` can modify pointers, which may not be the case for some functional, logic, or domain specific languages. Also, if the language has a construct `get_memory()` that returns the total memory size, which we can find in most real languages, and `code` requires exactly say 8GB of memory space as specified by ψ, then $\varphi * \varphi_{rest}\ \{\texttt{code}\}\ \psi * \varphi_{rest}$ does not hold for any nonempty φ_{rest}, so the rule is unsound. In other words, the (FRAME ON PROGRAMS) proof rule is a privilege of certain toy programming languages, or abstractions of real languages, whose soundness must be established for each language on a case by case basis. In contrast, (FRAME) in matching μ-logic is universally sound for all logical theories and thus programming languages whose semantics are defined as matching μ-logic theories. If one's particular language allows a proof rule like (Frame on programs), then one can prove it as a separate lemma and then use it in proofs.

6.1.4 Framework description

We present and discuss the automated proof rules in the framework, as shown in Figure 6.2. The framework is parametric in a theory Γ, and it proves implications, i.e., $\Gamma \vdash \varphi \to \psi$. A *proof rule* consists of several *premises* written above the bar and a *conclusion* written below the bar. Our prover takes the proposed proof rules and axioms in theory Γ and reduces the (given) proof goal by applying the rules backward, from conclusion to premises. New sub-goals will be generated during the proof. When all sub-goals are discharged, the prover stops with success. Therefore, our prover is essentially a simple search algorithm over the set of proof rules.

Before explaining the proof rules, we define some terminology. A *structure pattern* is a pattern built only from variables and symbols. A *conjunctive (resp. disjunctive) pattern* is a pattern of the form $\varphi_1 \land \cdots \land \varphi_n$ (resp. $\varphi_1 \lor \cdots \lor \varphi_n$), where $\varphi_1, \ldots, \varphi_n$ are structure patterns. In Figure 6.2, we assume p is a recursive symbol defined by $p(\tilde{x}) =_{\mathbf{lfp}} \bigvee_i \varphi_i$ where each φ_i denotes one definition case.

(elim-∃) is a standard FOL rule that simplifies the left-hand side by removing existential variables. Note that the side condition $x \notin \mathit{freeVar}(\psi)$ is necessary for the soundness of the rule, but it can be easily satisfied by renaming the bound variables to some fresh ones. Therefore, by applying (elim-∃) exhaustively, we can obtain a left-hand side that is quantifier-free at the top.

(smt) does domain reasoning using SMT solvers such as Z3 [97] and CVC4 [98], where recursive symbols are treated as uninterpreted functions. Note that (smt) is the only proof rule that finishes the proof, so it is always tried first. In practice, goals that can be proved by (smt) are those about the common mathematical domains such as natural and integer numbers, using the underlying theory Γ. We write $\vDash_{\mathtt{SMT}} \varphi \to \psi$ to mean that $\varphi \to \psi$ is proved by SMT solvers.

(pm) uses the pattern matching algorithm, `pm`, to instantiate the quantified variable(s) $\tilde{y}$ on the right-hand side. The algorithm `pm` will be discussed in Section 6.3.3. The algorithm returns a match result as a substitution θ, which tells us how to instantiate the variables $\tilde{y}$. If match succeeds, the instantiated proof goal $\varphi \to \psi\theta$ should be immediately proved by (smt).

Note that the soundness of our proof framework does not rely on the correctness of the matching algorithm, because (pm) is basically a standard FOL proof rule and holds for any

(ELIM-∃) $$\frac{\varphi \to \psi}{(\exists x \,.\, \varphi) \to \psi} \text{ if } x \notin \mathit{freeVar}(\psi)$$

(SMT) $$\varphi \to \psi \quad \text{if } \vDash_{\mathtt{SMT}} \varphi \to \psi$$

(MATCH-CTX) $$\frac{C_{rest}[\varphi'\theta] \to \psi}{C_o[\forall \tilde{y} \,.\, (C' \multimap \varphi')] \to \psi} \text{ where } (C_{rest}, \theta) = \mathsf{cm}(C_o, C', \tilde{y})$$

(PM) $$\frac{\varphi \to \psi\theta}{\varphi \to \exists \tilde{y} \,.\, \psi} \text{ where } \theta \in \mathtt{pm}(\varphi, \psi, \tilde{y}) \text{ matches } \varphi \text{ with } \psi$$

(FRAME) $$\frac{\varphi \to \psi}{C[\varphi] \to C[\psi]}$$

(UNFOLD-R) $$\frac{\varphi \to C[\varphi_i]}{\varphi \to C[p(\tilde{x})]} \text{ where } p(\tilde{x}) =_{\mathbf{lfp}} \bigvee_i \varphi_i$$

(KT) (the sequential compositions of the next 5 rules)

(WRAP) $$\frac{p(\tilde{x}) \to (C \multimap \psi)}{C[p(\tilde{x})] \to \psi}$$

(INTRO-∀) $$\frac{p(\tilde{x}) \to \forall \tilde{y} \,.\; (C \multimap \psi)}{p(\tilde{x}) \to (C \multimap \psi)} \text{ where } \tilde{y} = \mathit{freeVar}(\psi) \setminus \tilde{x}$$

(LFP) $$\frac{\cdots \;\; \varphi_i[\forall \tilde{y} \,.\, (C \multimap \psi)/p] \to \forall \tilde{y} \,.\, (C \multimap \psi)}{p(\tilde{x}) \to \forall \tilde{y} \,.\, (C \multimap \psi)}$$

(ELIM-∀) $$\frac{\varphi \to (C \multimap \psi)}{\varphi \to \forall y \,.\, (C \multimap \psi)} \text{ if } y \notin \mathit{freeVar}(\varphi)$$

(UNWRAP) $$\frac{C[\varphi] \to \psi}{\varphi \to (C \multimap \psi)}$$

Procedures `pm` and `cm` are defined in Sections 6.3.2 and 6.3.3.

Figure 6.2: Automatic Proof Rules for Fixpoint Reasoning

substitution θ. The matching algorithm is a heuristic to find a good θ. We rely on the external SMT solver to check the correctness of the match result given by the matching algorithm, through rule (SMT).

The combination of (PM) (based on the **p**attern **m**atching algorithm pm) and (SMT) (based on SMT solvers) gives us the ability to do static reasoning about structure patterns. In separation logic (SL), for example, structural patterns correspond to spatial formulas built from the heap constructors emp, $\mapsto$, and $*$, whose behaviors are axiomatized as the algebraic specification given in Section 2.6 where $*$ is associative and commutative and emp is its unit. If the matching algorithm pm does not support matching modulo associativity (A), commutativity (C), and unit elements (U), then it cannot effectively discharge (separation logic) goals that are provable. In general, matching modulo any (given) set of equations is undecidable [99], so in this paper, we implement a naive matching algorithm that supports matching modulo associativity (A-matching), and matching modulo associativity and commutativity (AC-matching), which turned out to be effective so far.

(UNFOLD-R) unfolds one recursive pattern $p(\tilde{x})$ on the right-hand side within any context C (satisfying mild conditions for contextual implication in Section 6.1.2) following its definition $p(\tilde{x}) =_{\mathbf{lfp}} \bigvee_i \varphi_i$. The technical conditions guarantee that disjunction distributes over the context, so $C[\bigvee_i \varphi_i] = \bigvee_i C[\varphi_i]$. Therefore, after applying (UNFOLD-R) we need to prove one of the new goals $\varphi \to C[\varphi_i]$.

(KT), named after the Knaster-Tarski fixpoint theorem [63], is a sequential composition of five proof rules shown in Figure 6.2: (WRAP), (INTRO-$\forall$), (LFP), (ELIM-$\forall$), and (UNWRAP). We explained the core proof rule (LFP) in Section 6.1.1. We explained in Section 6.1.2 why we need (WRAP) and (UNWRAP) and showed how they help address the limitations of (LFP), so here we only present their formal forms. (INTRO-$\forall$) and (ELIM-$\forall$) are standard FOL rules. (INTRO-$\forall$) strengthens the right-hand side and thus makes the subsequent proofs easier, because the (strengthened) right-hand side will be moved to the left-hand side by (LFP). Then after (LFP), we apply (ELIM-$\forall$) to restore the right-hand side to the form right after (WRAP) is applied (note the premise of (WRAP) is the same as the premise of (ELIM-$\forall$)).

There is a challenge raised by applying (LFP) on goals whose right-hand side are contextual implications, because those contextual implications are moved to the left-hand side by (LFP) and then block the proofs, because (so far) we have not defined any proof rules that can handle contextual implications on the left-hand side. This will be solved by (MATCH-CTX) which is explained below.

(MATCH-CTX) deals with the (quantified) contextual implication $\forall \tilde{y} \,.\, (C' \multimap \psi')$ on the left-hand side introduced by (LFP) and is one of the most complicated proof rule in our proof system. Note that (LFP) does the substitution $[\forall \tilde{y} \,.\, (C \multimap \psi)/p]$, which means (see

Section 6.1.1) to replace each recursive occurrence $p(\tilde{x}')$ (where $\tilde{x}'$ might be different from the original argument $\tilde{x}$) by $(\forall \tilde{y} \,.\, (C \multimap \psi))[\tilde{x}'/\tilde{x}]$, whose result we denote as $\forall \tilde{y} \,.\, (C' \multimap \psi')$. The number of contextual implications on the left-hand side is the same as the number of recursive occurrences of p in its definition. (MATCH-CTX) eliminates one contextual implication at a time, through a **c**ontext **m**atching algorithm cm, which will be discussed in Section 6.3.2. Here, we give the key intuition behind it.

When can a contextual implication $C' \multimap \psi'$ be eliminated? Recall Definition 6.1, which defines $C' \multimap \psi'$ to be the set of elements h such that $C'[h]$ satisfies ψ'. Therefore, we have the following key property about contextual implications:

$$\vdash C'[C' \multimap \psi'] \to \psi' \tag{6.4}$$

This property is not unexpected. Indeed, $C' \multimap \psi'$ is matched by any elements that imply ψ' when plugged in context C'. The above is a direct formalization of that intuition.

In principle, (6.4) can be used to handle contextual implication on the left-hand side. If contextual implication $C' \multimap \psi$ happens to occur within the same context C', then we can replace $C'[C' \multimap \psi]$ by ψ', using (6.4) and standard propositional reasoning. However, situations in practice are more complex. Firstly, contextual implication can be *quantified*, i.e., $\forall \tilde{y} \,.\, (C' \multimap \psi')$, so we need to first instantiate it using a substitution θ, to $C'\theta \multimap \psi'\theta$. Secondly, the out-most context C_o might contain more than needed to match with $C'\theta$. So after matching, the rest, unmatched context, denoted C_{rest}, stays in the proof goal. The **c**ontext **m**atching algorithm cm (Section 6.3.2) implements heuristics to find a suitable substitution θ such that $C'\theta$ matches with (a part of) the out-most context C_o, and when succeeding, it returns θ and the remaining unmatched context C_{rest}.

(FRAME) is to support frame reasoning. In contrast to (MATCH-CTX), which uses the outer context to simply the contextual implication, i.e. it says the context does matter, (FRAME) is to remove the outer context, which does not matter.

We conclude by the soundness of the proof rules in Figure 6.2.

Theorem 6.1. If φ is provable from Γ using the proof rules in Figure 6.2 then φ is provable from Γ using the proof system $\mathcal{H}_\mu$ in Figure 4.1 plus the proof rule (SMT).

Proof. (ELIM-∃), (PM), (INTRO-∀), (ELIM-∀) can be proved by standard FOL reasoning, which are supported by the proof system $\mathcal{H}_\mu$. Rules (LFP) and (UNFOLD-R) can be proved by standard fixpoint reasoning, also supported by $\mathcal{H}_\mu$. Rules (FRAME), (MATCH-CTX), (WRAP), and (UNWRAP) rely on the properties of structure contexts. QED.

Combining Theorem 6.1 with Theorem 4.1, we conclude that our proof framework is sound, assuming that the SMT solvers used in the proof rule (SMT) are sound.

Theorem 6.2. If φ is provable from Γ using the proof rules in Figure 6.2, then $\Gamma \vDash \varphi$, assuming the soundness of the SMT solvers used in the proof rule (SMT).

6.2 EXAMPLES

We have so far explained our proof rules. Next, we show how these rules are put into practice by using them to prove several example proof goals collected from the various logical systems. Our objective is to help the reader understand better our proof framework and some subtle technical details, to show that the proof rules in Figure 6.2 are designed carefully to capture the essence of fixpoint reasoning, and to show that our proof method is general and can be used to reason about fixpoints that occur in various mathematical domains.

6.2.1 A basic SL example

We first prove $\vdash \mathit{ll}(x,y) \to \mathit{lr}(x,y)$ where

$$\mathit{ll}(x,y) =_{\mathbf{lfp}} (x = y \land \mathsf{emp}) \lor (x \neq y \land \exists t \,.\, x \mapsto t * \mathit{ll}(t,y))$$
$$\mathit{lr}(x,y) =_{\mathbf{lfp}} (x = y \land \mathsf{emp}) \lor (x \neq y \land \exists t \,.\, \mathit{lr}(x,t) * t \mapsto y)$$

The proof tree is shown in Figure 6.3. Since the left-hand side $\mathit{ll}(x,y)$ is already a recursive pattern, the (WRAP) rule does not make any change. Therefore, we apply directly the (LFP) rule and get two new proof goals. One goal, shown below, corresponds to the base case of the definition of $\mathit{ll}(x,y)$:

$$\vdash (x = y \land \mathsf{emp}) \to \mathit{lr}(x,y)$$

The other goal corresponds to the inductive case and is shown in the second last line in Figure 6.3. For clarity, we breakdown the steps in calculating the substitution $[\mathit{lr}(x,y)/\mathit{ll}]$ required by (LFP) below:

$$\vdash \mathit{ll}(x,y) \to \mathit{lr}(x,y) \qquad \text{proof goal, before (LFP)}$$
$$\vdash (\exists z \,.\, x \mapsto z * \mathit{ll}(z,y) \land x \neq y) \to \mathit{lr}(x,y) \qquad \text{phantom step 1: unfolding}$$
$$\vdash (\exists z \,.\, x \mapsto z * \mathit{lr}(z,y) \land x \neq y) \to \mathit{lr}(x,y) \qquad \text{phantom step 2: substituting } \mathit{lr} \text{ for } \mathit{ll}$$

Now, the base case goal can be proved by applying (UNFOLD-R) to unfold the right-hand side $\mathit{lr}(x,y)$ to its base case and then calling SMT solvers. The inductive case (after eliminating

$$
\textsc{lfp}\;\dfrac{
  \textsc{elim-}\exists\;\dfrac{
    \textsc{wrap}\;\dfrac{
      \textsc{intro-}\forall\;\dfrac{
        \textsc{lfp}\;\dfrac{
          \textsc{elim-}\forall\;\dfrac{
            \textsc{unwrap}\;\dfrac{
              \textsc{elim-}\exists\;\dfrac{
                \textsc{match-ctx}\;\dfrac{
                  \textsc{unfold-r}\;\dfrac{
                    \textsc{pm}\;\dfrac{
                      \textsc{smt}\;\dfrac{\mathtt{True}}{lr(x,w) * w \mapsto y \land z \neq y \land x \neq y \to x \neq y \land lr(x,w) * w \mapsto y}
                    }{lr(x,w) * w \mapsto y \land z \neq y \land x \neq y \to x \neq y \land \exists t\,.\,lr(x,t) * t \mapsto y}
                  }{lr(x,w) * w \mapsto y \land z \neq y \land x \neq y \to lr(x,y)}
                }{x \mapsto z * (\forall x\,.\,(C' \multimap lr(x,w))) * w \mapsto y \land z \neq y \land x \neq y \to lr(x,y) \qquad (\dagger)}
              }{\exists w\,.\,x \mapsto z * (\forall x\,.\,(C' \multimap lr(x,w))) * w \mapsto y \land z \neq y \land x \neq y \to lr(x,y)}
            }{\exists w\,.\,(\forall x\,.\,(C' \multimap lr(x,w))) * w \mapsto y \land z \neq y \to (C \multimap lr(x,y))}
          }{\exists w\,.\,(\forall x\,.\,(C' \multimap lr(x,w))) * w \mapsto y \land z \neq y \to \forall x\,.\,(C \multimap lr(x,y))}
          \qquad \cdots
        }{lr(z,y) \to \forall x\,.\,(C \multimap lr(x,y))}
      }{lr(z,y) \to (C \multimap lr(x,y))}
    }{x \mapsto z * lr(z,y) \land x \neq y \to lr(x,y)}
  }{\exists z\,.\,x \mapsto z * lr(z,y) \land x \neq y \to lr(x,y)}
  \qquad \cdots
}{ll(x,y) \to lr(x,y)}
$$

$$
\text{where}\quad
\begin{aligned}
C[\square] &\equiv x \mapsto z * \square \land x \neq y \\
C'[\square] &\equiv x \mapsto z * h \land x \neq w
\end{aligned}
$$

Figure 6.3: Proof Tree of $\vdash ll(x,y) \to lr(x,y)$

$\exists z$ from left-hand side), $\vdash x \mapsto z * lr(z,y) \land x \neq y \to lr(x,y)$, contains a recursive pattern $lr(z,y)$ within a context $C[h] = x \mapsto z * h \land x \neq y$. Therefore, we (WRAP) the context and yield contextual implication $C \multimap lr(x,y)$ on the right-hand side, and quantify it with $\forall x$ by (INTRO-$\forall$). Then (LFP) is applied, yielding two sub-goals, one for the base case and one for the inductive case. We omit the base case and show the following breakdown steps for the inductive case, for clarity:

$$
\begin{aligned}
&\vdash lr(z,y) \to (C \multimap lr(x,y)) && \text{proof goal, before (LFP)}\\
&\vdash (\exists w\,.\,lr(z,w) * w \mapsto y \land z \neq y) \to (C \multimap lr(x,y)) && \text{unfolding}\\
&\vdash (\exists w\,.\,(\forall x\,.\,(C \multimap lr(x,y)))[w/y]{*}w{\mapsto}y{\land}z \neq y){\to}(C \multimap lr(x,y)) && \text{substituting}
\end{aligned}
$$

where $(\forall x\,.\,(C \multimap lr(x,y)))[w/y] = \forall x\,.\,(C' \multimap lr(x,w))$ and $C'[h] = x \mapsto z * h \land x \neq w$.

Now the proof proceeds by (UNWRAP)-ping the context C on the right-hand side and moving it back to the left-hand side, and eliminating the quantifier $\exists w$ by (ELIM-$\exists$). Then the proof goal becomes the following (i.e., ($\dagger$) in line 5, Figure 6.3):

$$x \mapsto z * (\forall x\,.\,(C' \multimap lr(x,w))) * w \mapsto y \land z \neq y \land x \neq y \to lr(x,y)$$

At this point, the quantified contextual implication on the left-hand side is instantiated and matched by (MATCH-CTX), which calls the context matching algorithm cm, introduced in Section 6.3. Intuitively, the algorithm uses heuristics to produce an instantiation for $\forall x$ (in

$$
\text{LFP}\;\dfrac{
\text{ELIM-}\exists\;\dfrac{
\text{WRAP}\;\dfrac{
\text{INTRO-}\forall\;\dfrac{
\text{LFP}\;\dfrac{
\text{ELIM-}\forall\;\dfrac{
\text{UNWRAP}\;\dfrac{
\text{ELIM-}\exists\;\dfrac{
\text{MATCH-CTX}\;\dfrac{
\text{UNFOLD-R}\;\dfrac{
\text{PM}\;\dfrac{
\text{SMT}\;\dfrac{\texttt{True}}{lr(x,w,s_3)*\phi \to lr(x,w,s_3)*w\mapsto y\wedge s{=}s_3{\cup}\{w\}\wedge x\neq y}
}{lr(x,w,s_3)*\phi\to\exists t\exists s_4\,.\,lr(x,t,s_4)*t\mapsto y\wedge s{=}s_4{\cup}\{t\}\wedge x\neq y}
}{lr(x,w,s_3)*\phi\to lr(x,y,s)}
}{x\mapsto z*(\forall x\forall s\,.\,(C'\multimap lr(x,w,s)))*\phi\to lr(x,y,s)\qquad(\ddagger)}
}{x\mapsto z*(\exists w\exists s_2\,.\,(\forall x\forall s\,.\,(C'\multimap lr(x,w,s)))*\phi)\wedge s{=}s_1{\cup}\{x\}\wedge x\neq y\to lr(x,y,s)}
}{\exists w\exists s_2\,.\,(\forall x\forall s\,.\,(C'\multimap lr(x,w,s)))*w\mapsto y\wedge s_1{=}s_2{\cup}\{w\}\wedge z\neq y\to(C\multimap lr(x,y,s))}
}{\exists w\exists s_2\,.\,(\forall x\forall s\,.\,(C'\multimap lr(x,w,s)))*w\mapsto y\wedge s_1{=}s_2{\cup}\{w\}\wedge z\neq y\to\forall x\forall s\,.\,(C\multimap lr(x,y,s))\qquad\cdots}
}{lr(z,y,s_1)\to\forall x\forall s\,.\,(C\multimap lr(x,y,s))}
}{lr(z,y,s_1)\to(C\multimap lr(x,y,s))}
}{x\mapsto z*lr(z,y,s_1)\wedge s{=}s_1{\cup}\{x\}\wedge x\neq y\to lr(x,y,s)}
}{\exists z\exists s_1\,.\,x\mapsto z*lr(z,y,s_1)\wedge s{=}s_1{\cup}\{x\}\wedge x\neq y\to lr(x,y,s)\qquad\cdots}
}{ll(x,y,s)\to lr(x,y,s)}
$$

$$
\begin{aligned}
\text{where}\quad & C[\square]\equiv x\mapsto z*\square\wedge s=s_1\cup\{x\}\wedge x\neq y\\
& C'[\square]\equiv C[\square][w/y,s_2/s_1]=x\mapsto z*\square\wedge s=s_2\cup\{x\}\wedge x\neq w\\
& \phi\equiv w\mapsto y\wedge s_1=s_2\cup\{w\}\wedge z\neq y\wedge s=s_1\cup\{x\}\wedge x\neq y
\end{aligned}
$$

Figure 6.4: Proof Tree of $\vdash ll(x,y,s)\to lr(x,y,s)$

this case, it happens that the algorithm instantiates $\forall x$ to x) and then checks if the out-most context C_o of (†) implies the (instantiated) context C', where $C_o[h]\equiv x\mapsto z*h*w\mapsto y\wedge z\neq y\wedge x\neq y$.

Note that context C' consists of a structure pattern $x\mapsto z$ and a logical constraint $x\neq w$. The structure pattern is already matched in C_o. The logical constraint can be implied from C_o, which has two structure patterns $x\mapsto z$ and $w\mapsto y$, and using the basic SL axiom/property $x_1\mapsto y*x_2\mapsto z\to x_1\neq x_2$. Therefore, (MATCH-CTX) is applied successfully, and the rest, unmatched context of C_o is left in the goal (line 4 of Figure 6.3) and proved in the subsequent proofs.

6.2.2 A more complex SL example

The previous simple example does not illustrate the usage of (INTRO-∀), because (MATCH-CTX) applied to goal (†) in Figure 6.3 decides to instantiate $\forall x$ by x, which means that the proof could also work without (INTRO-∀). In this section, we show a slightly more complex example that shows the necessity of (INTRO-∀).

Consider the following slightly modified definitions of ll and lr that take a third argument s denoting the set of elements in the list segment:

$$ll(x,y,s)=_{\mathbf{lfp}}(x=y\wedge\mathsf{emp}\wedge s=\emptyset)\vee\exists x_1\exists s_1\,.\,x{\mapsto}x_1*ll(x_1,y,s_1)\wedge s{=}s_1{\cup}\{x\}\wedge x{\neq}y$$

$$\textsc{wrap}\;\dfrac{\textsc{intro-}\forall\;\dfrac{\textsc{lfp}\;\dfrac{\textsc{elim-}\forall\;\dfrac{\textsc{unwrap}\;\dfrac{\textsc{elim-}\exists\;\dfrac{\textsc{match-ctx}\;\dfrac{\textsc{unfold-r}\;\dfrac{\textsc{pm}\;\dfrac{\textsc{smt}\;\dfrac{\mathsf{True}}{x \mapsto t_1 * t_1 \mapsto t_2 * llE(t_2,z) \rightarrow x \mapsto t_1 * t_1 \mapsto t_2 * llE(t_2,z)}}{x \mapsto t_1 * t_1 \mapsto t_2 * llE(t_2,z) \rightarrow \exists u_1 \exists u_2 \,.\, x \mapsto u_1 * u_1 \mapsto u_2 * llE(u_2,z)}}{x \mapsto t_1 * t_1 \mapsto t_2 * llE(t_2,z) \rightarrow llE(x,z)}}{x \mapsto t_1 * t_1 \mapsto t_2 * (\forall z \,.\, (C \multimap llE(t_2,z))) * llO(y,z) \rightarrow llE(x,z)}}{\exists t_1 \exists t_2 \,.\, x \mapsto t_1 * t_1 \mapsto t_2 * (\forall z \,.\, (C \multimap llE(t_2,z))) * llO(y,z) \rightarrow llE(x,z)}}{\exists t_1 \exists t_2 \,.\, x \mapsto t_1 * t_1 \mapsto t_2 * \forall z \,.\, (C \multimap llE(t_2,z)) \rightarrow (C \multimap llE(x,z))}}{\exists t_1 \exists t_2 \,.\, x \mapsto t_1 * t_1 \mapsto t_2 * \forall z \,.\, (C \multimap llE(t_2,z)) \rightarrow (C \multimap llE(x,z))} \qquad \cdots}{llO(x,y) \rightarrow \forall z \,.\, (C \multimap llE(x,z))}}{llO(x,y) \rightarrow (C \multimap llE(x,z))}}{llO(x,y) * llO(y,z) \rightarrow llE(x,z)}$$

$$\text{where} \quad C[\square] \equiv \square * llO(y,z)$$

Figure 6.5: Proof Tree of $\vdash llO(x,y) * llO(y,z) \rightarrow llE(x,z)$

$$lr(x,y,s) =_{\mathbf{lfp}} (x = y \wedge \mathsf{emp} \wedge s = \emptyset) \vee \exists y_1 \exists s_1 \,.\, lr(x,y_1,s_1) * y_1 {\mapsto} y \wedge s{=}s_1{\cup}\{y_1\} \wedge x{\neq}y$$

Its proof tree in Figure 6.4 is similar to the one in Figure 6.3, except that the use of rule (INTRO-∀) is *necessary* for the proof to succeed, because we need to *instantiate* the quantifier $\forall s$ of goal (‡) in Figure 6.4, line 5, with a fresh variable s_3 in the application of rule (MATCH-CTX). Suppose there is no application of rule (INTRO-∀). Then, we will have

$$x \mapsto z * (C' \multimap lr(x,w,s)) * w \mapsto y \wedge s_1 = s_2 \cup \{w\} \wedge z \neq y \wedge s = s_1 \cup \{x\} \wedge x \neq y \rightarrow lr(x,y,s)$$

where $C'[\square] = x \mapsto z * \square \wedge s = s_2 \cup \{x\} \wedge x \neq w$. So we cannot match $s{=}s_1{\cup}\{x\} \wedge s_1{=}s_2{\cup}\{w\}$ in the outer context with $s{=}s_2{\cup}\{x\}$ in the inner context. In other words, we cannot eliminate the inner context and the proof will get stuck.

6.2.3 A SL example featuring mutual recursion

Mutually recursive definitions are in general defined as:

$$\begin{cases} p_1(\tilde{y}_1) =_{\mathbf{lfp}} \exists \tilde{x}_{11} \,.\, \varphi_{11}(\tilde{y}_1, \tilde{x}_{11}) \vee \cdots \vee \exists \tilde{x}_{1m_1} \,.\, \varphi_{1m_1}(\tilde{y}_1, \tilde{x}_{1m_1}) \\ \qquad \cdots \\ p_k(\tilde{y}_k) =_{\mathbf{lfp}} \exists \tilde{x}_{k1} \,.\, \varphi_{k1}(\tilde{y}_k, \tilde{x}_{k1}) \vee \cdots \vee \exists \tilde{x}_{km_k} \,.\, \varphi_{km_k}(\tilde{y}_k, \tilde{x}_{km_k}) \end{cases}$$

which simultaneously define k recursive definitions $p_1, \ldots, p_k$ to be the least among those satisfy the equations. Our way of dealing with mutual recursion is to reduce it to several non-mutual, simple recursions. We use the following separation logic challenge test

$$\textsc{gfp}\ \dfrac{\textsc{pm}\ \dfrac{\circ\wedge\ \dfrac{\textsc{unfold-L}\ \dfrac{\textsc{smt}\ \dfrac{\texttt{True}}{p \wedge (p \to \circ p) \wedge \circ\Box(p \to \circ p) \to \circ p \wedge \circ\Box(p \to \circ p)}}{p \wedge \Box(p \to \circ p) \to \circ p \wedge \circ\Box(p \to \circ p)}}{p \wedge \Box(p \to \circ p) \to \circ(p \wedge \Box(p \to \circ p))}}{p \wedge \Box(p \to \circ p) \to p \wedge \circ(p \wedge \Box(p \to \circ p))}}{p \wedge \Box(p \to \circ p) \to \Box p}$$

Figure 6.6: Proof Tree of $\vdash p \wedge \Box(p \to \circ p) \to \Box p$

`qf_shid_entl/10.tst.smt2` from the SL-COMP'19 competition [96] as an example. Consider the following definition of list segments of odd and even length:

$$\begin{cases} llO(x,y) =_{\mathbf{lfp}} x \mapsto y \vee \exists t \,.\, x \mapsto t * llE(t,y) \\ llE(x,y) =_{\mathbf{lfp}} \exists t \,.\, x \mapsto t * llO(t,y) \end{cases}$$

and the proof goal $\vdash llO(x,y) * llO(y,z) \to llE(x,z)$.

To proceed the proof, we first reduce the mutual recursion definition into the following two non-mutual, simple recursion definitions, which can be obtained systematically by unfolding the other recursive symbols to exhaustion.

$$llO(x,y) =_{\mathbf{lfp}} x \mapsto y \vee \exists t_1 \exists t_2 \,.\, x \mapsto t_1 * t_1 \mapsto t_2 * llO(t_2,y)$$
$$llE(x,y) =_{\mathbf{lfp}} \exists t_1 \exists t_2 \,.\, x \mapsto t_1 * t_1 \mapsto t_2 * llE(t_2,y)$$

Then, the proof can be carried out in the normal way. We show the proof tree in Figure 6.5.

6.2.4 An LTL example

We demonstrate the generality of our proof method by showing how to prove the induction proof rule of the sound and complete proof system of LTL (Figure 2.5. Recall that LTL can be defined as a matching μ-logic theory (Section 5.9.1).

Consider the following LTL rule for induction: $\vdash p \wedge \Box(p \to \circ p) \to \Box p$. Since the "always $\Box$" operator is defined as a greatest fixpoint $\Box\varphi =_{\mathbf{gfp}} \varphi \wedge \circ\Box\varphi$, we need a set of proof rules dual to those in Figure 6.2, where the key rule, (GFP) (dual to (LFP)), is shown below:

$$(\textsc{gfp}) \quad \frac{\varphi \to \psi_i[\varphi/q]}{\varphi \to q(\tilde{y})} \qquad q(\tilde{y}) =_{\mathbf{gfp}} \bigvee \psi_i$$

(GFP) is used to discharge the right-hand side $\Box p$ of the proof goal. We show the selfexplanatory proof tree in Figure 6.6. Note that during the proof we use the distributivity law provided by the theory Γ^{LTL} in Section 5.9.1, denoted as proof step $(\circ\wedge)$ in Figure 6.6.

6.2.5 A verification example from RL

We have discussed RL and showed its matching μ-logic theory in Section 5.11. Here, we use one example to illustrate how reachability reasoning, i.e. formal verification, can be handled uniformly by our proof framework. Before we dive into the technical details, let us remind readers that in RL, structure patterns are used to represent the program states, called configurations in RL, of the programming language (Section 2.14). The reachability property $\varphi_1 \Rightarrow \varphi_2$ then builds on top of the structure patterns and defines the transition relation among program configurations.

We use the following simple program `sum` to explain the core RL concepts.

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

The program `sum` is written in a simple imperative language that has a C-like syntax. It calculates the total from 1 to n and adds it to the variable `s`. Its functional correctness means that when it terminates, the value of variable `s` should be $s + n(n-1)/2$, where s and n are the initial values we give to the variables `s` and `n`, respectively.

In order to execute `sum`, we need to know the concrete values of `s` and `n`. This semantic information is organized as a mapping from variables to their values and we call the mapping a state. Knowing the program and the state where it is executed allows us to execute the program to termination. Thus, a program and a state forms a complete computation configuration for this simple imperative language and the configurations can be represented using structure patterns that hold all the semantic information needed for program execution. For example, let us write down the initial and final configurations of `sum` where we initialize `s` and `n` by the integer values s and n, respectively:

$$\varphi_{pre} \equiv \big\langle \langle \texttt{sum} \rangle_{\texttt{code}} \langle \mathrm{n} \mapsto n,\ \mathrm{s} \mapsto s \rangle_{\texttt{state}} \big\rangle_{\texttt{cfg}} \ \wedge\ n \geq 1$$
$$\varphi_{post} \equiv \big\langle \langle \cdot \rangle_{\texttt{code}} \langle \mathrm{n} \mapsto 0, \mathrm{s} \mapsto s + n(n-1)/2 \rangle_{\texttt{state}} \big\rangle_{\texttt{cfg}}$$

Following RL convention, we write configurations in cells such as $\langle \ldots \rangle_{\texttt{code}}, \langle \ldots \rangle_{\texttt{state}}$; from a logical point of view, these are simply structure patterns and are built from ML symbols in the same way how FOL terms are defined. The functional correctness of `sum` states the following: if we start from the initial configuration φ_{pre} and the program terminates, then the final configuration is φ_{post}, where there is nothing to be executed anymore (as denoted by the

$$
{\scriptstyle\text{REACH}}\ \dfrac{{\scriptstyle\text{SYM}}\ \dfrac{{\scriptstyle\text{SYM}}\ \dfrac{{\scriptstyle\text{SYM}}\ \dfrac{{\scriptstyle\text{SYM}}\ \dfrac{{\scriptstyle\text{WRAP}}\ \dfrac{{\scriptstyle\text{INTRO-}\forall}\ \dfrac{{\scriptstyle\text{LFP}}\ \dfrac{{\scriptstyle\text{ELIM-}\forall}\ \dfrac{{\scriptstyle\text{UNWRAP}}\ \dfrac{{\scriptstyle\text{APP-SYM}}\ \dfrac{{\scriptstyle\text{FOL}}\ \dfrac{{\scriptstyle\text{UNFOLD-R}}\ \dfrac{{\scriptstyle\text{FRAME}}\ \dfrac{{\scriptstyle\text{PM}}\ \dfrac{{\scriptstyle\text{SMT}}\ \dfrac{S \vdash \texttt{True}}{S \vdash \varphi_3' \to \varphi_3'}}{S \vdash \varphi_3' \to \varphi_{fin}}}{S \vdash \bullet^3\varphi_3' \to \bullet^3\varphi_{fin}}}{S \vdash \bullet^3\varphi_3' \to \diamond\psi}}{S{\vdash}\circ^4(\forall n_1\forall s_1 \,.\, (C \multimap \varphi_{fin})){\land}\bullet^3\varphi_3'{\to}\varphi_{fin}} \quad {\scriptstyle\circ\bullet}\ \dfrac{{\scriptstyle\text{UNFOLD-R}}\ \dfrac{{\scriptstyle\text{FRAME}}\ \dfrac{{\scriptstyle\text{MATCH-CTX}}\ \dfrac{{\scriptstyle\text{PM}}\ \dfrac{{\scriptstyle\text{UNFOLD-R}}\ \dfrac{{\scriptstyle\text{PM}}\ \dfrac{{\scriptstyle\text{SMT}}\ \dfrac{S \vdash \texttt{True}}{S{\vdash}\textsc{sum}(1,n_1',s_1',s_2){\to}s_1'{=}s_1{+}n_1'{\land}n_1'{=}n_1{-}1{\land}\textsc{sum}(1,n_1',s_1',s_2)}}{S{\vdash}\textsc{sum}(1,n_1',s_1',s_2){\to}\exists x\exists y \,.\, y{=}s_1{+}x{\land}x{=}n_1{-}1{\land}\textsc{sum}(1,x,y,s_2)}}{S \vdash \textsc{sum}(1,n_1',s_1',s_2) \to \textsc{sum}(1,n_1,s_1,s_2)}}{S \vdash \Diamond\psi' \to \varphi_{fin}}}{S \vdash \forall n_1\forall s_1 \,.\, (C \multimap \varphi_{fin}) \land \varphi_4 \to \varphi_{fin}}}{S \vdash \bullet^4(\forall n_1\forall s_1 \,.\, (C \multimap \varphi_{fin}) \land \varphi_4) \to \bullet^4 \diamond \psi}}{S \vdash \bullet^4(\forall n_1\forall s_1 \,.\, (C \multimap \varphi_{fin}) \land \varphi_4) \to \varphi_{fin}}}{S \vdash \circ^4(\forall n_1\forall s_1 \,.\, (C \multimap \varphi_{fin})) \land \bullet^4\varphi_4 \to \varphi_{fin}}}{S \cup \{\varphi_{pre} \to (\bullet^3\varphi_3' \lor \bullet^4\varphi_4)\} \vdash \circ^4(\forall n_1\forall s_1 \,.\, (C \multimap \varphi_{fin})) \land \varphi_{pre} \to \varphi_{fin}}}{S \cup \{\varphi_{pre} \to (\bullet^3\varphi_3' \lor \bullet^4\varphi_4)\} \vdash \circ^4(\forall n_1\forall s_1 \,.\, (C \multimap \varphi_{fin})) \to (C \multimap \varphi_{fin})}}{S \cup \{\varphi_{pre} \to (\bullet^3\varphi_3' \lor \bullet^4\varphi_4)\} \vdash \circ^4(\forall n_1\forall s_1 \,.\, (C \multimap \varphi_{fin})) \to \forall n_1\forall s_1 \,.\, (C \multimap \varphi_{fin})}}{S \cup \{\varphi_{pre} \to (\bullet^3\varphi_3' \lor \bullet^4\varphi_4)\} \vdash \mu f.\, \circ^4 f \to \forall n_1\forall s_1 \,.\, (C \multimap \varphi_{fin})}}{S \cup \{\varphi_{pre} \to (\bullet^3\varphi_3' \lor \bullet^4\varphi_4)\} \vdash \mu f.\, \circ^4 f \to (C \multimap \varphi_{fin})}}{S \cup \{\varphi_{pre} \to (\bullet^3\varphi_3' \lor \bullet^4\varphi_4)\} \vdash \varphi_{pre} \to (\mu f.\, \circ^4 f \to \varphi_{fin})}}{S \cup \{\varphi_{pre} \to (\bullet^3\varphi_3' \lor \bullet^3\varphi_3)\} \vdash \varphi_{pre} \to (\mu f.\, \circ^3 f \to \varphi_{fin})}}{S \cup \{\varphi_{pre} \to \bullet^2(\varphi_2' \lor \varphi_2)\} \vdash \varphi_{pre} \to (\mu f.\, \circ^2 f \to \varphi_{fin})}}{S \cup \{\varphi_{pre} \to \bullet\varphi_1\} \vdash \varphi_{pre} \to (\mu f.\, \circ f \to \varphi_{fin})}}{S \cup \{\varphi_{pre} \to \varphi_{pre}\} \vdash \varphi_{pre} \to (\mu f.\, \circ f \to \varphi_{fin})}}{S \cup \{\varphi_{pre} \to \varphi_{pre}\} \vdash \varphi_{pre} \Rightarrow \psi}
$$

$$
\begin{array}{ll}
\textsc{sum}(l,u,b,s) =_{\mathbf{lfp}} (l{>}u{\land}s{=}b){\lor}(\exists b_1\exists u_1 \,.\, b_1{=}b{+}u_1{\land}u_1{=}u{-}1{\land}\textsc{sum}(l,u_1,b_1,s)) & \\
\psi \equiv \exists n_2\exists s_2 \,.\, \langle\langle\cdot\rangle_{\mathtt{code}}\langle \mathtt{n}{\mapsto}n_2,\ \mathtt{s}{\mapsto}s_2\rangle_{\mathtt{state}}\rangle_{\mathtt{cfg}} \land n_2 = 0 \land \textsc{sum}(1,n_1,s_1,s_2) & \\
\psi' \equiv \exists n_2\exists s_2 \,.\, \langle\langle\cdot\rangle_{\mathtt{code}}\langle \mathtt{n}{\mapsto}n_2,\ \mathtt{s}{\mapsto}s_2\rangle_{\mathtt{state}}\rangle_{\mathtt{cfg}} \land n_2 = 0 \land \textsc{sum}(1,n_1',s_1',s_2) & \\
\varphi_1 \equiv \langle\langle \mathtt{n--;\,cond}\rangle_{\mathtt{code}}\langle \mathtt{n}{\mapsto}n_1,\ \mathtt{s}{\mapsto}s_1\rangle_{\mathtt{state}}\rangle_{\mathtt{cfg}} \land n_1 \geq 1 & \mathtt{sum} \equiv \mathtt{while(-n)\{s=s+n;\}} \\
\varphi_2 \equiv \langle\langle \mathtt{cond}\rangle_{\mathtt{code}}\langle \mathtt{n}{\mapsto}n_1',\ \mathtt{s}{\mapsto}s_1\rangle_{\mathtt{state}}\rangle_{\mathtt{cfg}} \land n_1 \geq 2 & \mathtt{cond} \equiv \mathtt{if(n>0)\{s=s+n;sum\}} \\
\varphi_2' \equiv \langle\langle \mathtt{cond}\rangle_{\mathtt{code}}\langle \mathtt{n}{\mapsto}n_1',\ \mathtt{s}{\mapsto}s_1\rangle_{\mathtt{state}}\rangle_{\mathtt{cfg}} \land n_1 = 1 & \mathtt{body} \equiv \mathtt{s=s+n;sum} \\
\varphi_3 \equiv \langle\langle \mathtt{body}\rangle_{\mathtt{code}}\langle \mathtt{n}{\mapsto}n_1',\ \mathtt{s}{\mapsto}s_1\rangle_{\mathtt{state}}\rangle_{\mathtt{cfg}} \land n_1 \geq 1 & C[\square] \equiv \varphi_{pre} \land h \\
\varphi_3' \equiv \langle\langle\cdot\rangle_{\mathtt{code}}\langle \mathtt{n}{\mapsto}n_1',\ \mathtt{s}{\mapsto}s_1\rangle_{\mathtt{state}}\rangle_{\mathtt{cfg}} \land n_1 \geq 1 & n_1' \equiv n_1 - 1 \\
\varphi_4 \equiv \langle\langle \mathtt{sum}\rangle_{\mathtt{code}}\langle \mathtt{n}{\mapsto}n_1',\ \mathtt{s}{\mapsto}s_1'\rangle_{\mathtt{state}}\rangle_{\mathtt{cfg}} \land n_1 \geq 1 & s_1' \equiv s_1 + n_1 - 1
\end{array}
$$

Figure 6.7: Verifying Functional Correctness of `sum` using Reachability Rules

dot “ · ”, meaning “nothing”, in the $\langle\ldots\rangle_{\mathtt{code}}$ cell), `n` is mapped to 0, and `s` is mapped to the correct total $s + n(n-1)/2$. This functional (partial) correctness property can be expressed by the reachability property $\varphi_{pre} \Rightarrow \varphi_{post}$. According to Section 5.11, $\varphi_{pre} \Rightarrow \varphi_{post}$ is equal to $\varphi_{pre} \to (\mathsf{WF} \to \diamond\varphi_{post})$, where $\mathsf{WF} = \mu X \,.\, \circ X$ is matched by all well-founded configurations (i.e., those without infinite execution traces) and $\diamond\varphi_{post} = \mu X \,.\, \varphi_{\text{post}} \lor \bullet X$ is matched by all configurations that eventually reach φ_{post}, after at most finitely many execution steps. This encoding correctly captures the partial correctness.

We now prove that `sum` satisfies the correctness property $\varphi_{pre} \Rightarrow \varphi_{post}$. We put the proof tree in Figure 6.7 and explain it at a higher-level below. Intuitively, the proof works by symbolically executing the program step by step and applying inductive reasoning to finish the proof as soon as repetitive configurations (i.e., those generated by the while-loop in `sum`) are identified during the proof. Each symbolic execution step corresponds to a reachability property that can be proved about `sum`. While we proceed with the proof and carry out symbolic execution, we collect the proved reachability properties so that they can be used (by induction) to resolve the proof goal about the while-loop.

The proof goals have the form $S \cup S_i \vdash \varphi_i \to \psi_i$ where S is a set of RL rules that include all the reachability rules axiomatizing the small-step style operational semantics of the language and S_i include those representing the results of i-step symbolic execution. Initially, the functional correctness proof goal is $S \cup \{\varphi_{pre} \to \varphi_{pre}\} \vdash \varphi_{pre} \to \diamond_w \psi$, where ψ is the final configuration φ_{post} rewritten using the recursive predicate $\textsc{sum}(l, u, b, s)$, meaning the partial-sum relation: $s = b + (u + (u - 1) + \cdots + l)$. Pattern $\varphi_{pre} \to \varphi_{pre}$ corresponds to the symbolic execution reachability rule (i.e., lemma) that we can prove by executing the initial configuration φ_{pre} by 0 step. As the proof proceeds, more symbolic execution steps are carried out and more lemmas are proved. The following domain-specific rule is used to carry out symbolic execution and flush the newly-proved lemmas/rules that summarize the semantics of SUM into S_i:

$$(\textsc{sym})\,\frac{S \cup S_k \vdash \varphi \to (\mu f.\, \circ^j f \to \Diamond\psi)}{S \cup \{\varphi \to \varphi'\} \vdash \varphi \to (\mu f.\, \circ^i f \to \Diamond\psi)} \qquad \text{if } S_k \neq \emptyset \wedge i \geq 1 \text{ where } (S_k, j) = \textsc{next}(\varphi')$$

where $\textsc{next}$ takes the current symbolic configuration, executes it according to the semantics S, and outputs a rule that specifies the step (implemented similarly to [3]) and the number of steps taken. We stop execution when the code cell $\langle \ldots \rangle_{\mathsf{code}}$ becomes empty (as in the case of φ_3') or contains the same code as that of φ_{pre} (as in the case of φ_4). The collected rules (e.g. $\{\varphi_{pre} \to (\bullet^3 \varphi_3' \vee \bullet^4 \varphi_4)\}$) will be used to simplify φ_{pre} later (e.g. as in the application of ($\textsc{app-sym}$)).

6.3 ALGORITHMS

Our generic matching μ-logic prover runs a simple DFS algorithm on top of the proof rules in Figure 6.2. In this section, we show the top-level DFS algorithm in Figure 6.8. We also show the pattern matching algorithms used by the proof rules ($\textsc{pm}$) and ($\textsc{match-ctx}$) in Figure 6.9.

6.3.1 Top-level DFS proof search algorithm

The top-level proof search algorithm in Figure 6.8 starts with procedure Prove on the goal $\vdash \varphi \to \psi$, which uses two counters $c_{\textsc{ru}}, c_{\textsc{kt}}$, both initialized to zero, to keep track of how many times ($\textsc{Unfold-r}$) and ($\textsc{kt}$) have been applied. Proof search terminates (unsuccessfully) if

```
function Prove($\varphi \to \psi,\ c_{\text{RU}},\ c_{\text{KT}}$)
    if (BasicProof($\varphi \to \psi$)) return $true$
    let $\{p_i\}$:=rec_sym($\varphi$), $\{q_i\}$:=rec_sym($\psi$), $OrSet := \emptyset$
    foreach $(\forall\tilde{y} \,.\, C' \multimap \varphi') \in \varphi$    /* (MATCH-CTX)*/
        $C_0 := \varphi \setminus \{\forall\tilde{y} \,.\, C' \multimap \varphi'\}$
        $(C_{rest}, \theta) := \mathsf{cm}(C_0, C', \tilde{y})$    /*Section 6.3.2*/
        $\varphi'' := C_{rest} \cup \varphi'\theta$
        $ob := [\varphi'' \to \psi,\ c_{\text{RU}},\ c_{\text{KT}}]$,   $OrSet \ {}_{\cup}{=} \{\{ob\}\}$
    foreach $p_i \in \varphi, q_{i'} \in \psi$    /* (FRAME)*/
        if ($(\theta := \mathtt{pm}([p_i], [q_{i'}], freeVar(\psi) \setminus freeVar(\varphi))) \neq$ Failure)
            $\varphi' := \varphi \setminus \{p_i\}, \psi' := (\psi \setminus \{q_{i'}\})\theta$
            $ob := [\varphi' \to \psi',\ c_{\text{RU}},\ c_{\text{KT}}]$,   $OrSet \ {}_{\cup}{=} \{\{ob\}\}$
    if ($c_{\text{RU}} <$ MAXRIGHTBOUND)    /* (UNFOLD-R)*/
        foreach ($q_i \in \psi$)
            foreach ($\psi_j \in (\{\psi_1 \ldots \psi_k\} :=$ UNFOLD($\psi, q_i$)))
                $ob := [\varphi \to \psi_j, c_{\text{RU}} + 1, c_{\text{KT}}]$,  $OrSet \ {}_{\cup}{=} \{\{ob\}\}$
    if ($c_{\text{KT}} <$ KTBOUND)    /* (KT) */
        foreach ($p_i \in \varphi$)
            foreach ($\varphi_j \in (\{\varphi_1, \varphi_2, \ldots \varphi_l\} :=$ KT($\varphi, p_i$)))
                $ob := [\varphi_j \to \psi, c_{\text{RU}}, c_{\text{KT}} + 1]$
                if (trivially_true($ob$)) continue
                $Obs := Obs \cup \{ob\}$
            $OrSet \ {}_{\cup}{=} \{Obs\}$
    if ($OrSet = \emptyset$) return $false$
    $OrSet :=$ OrderByHeuristics($OrSet$)
    foreach ($Obs \in OrSet$)
        if (ProveAll($Obs$)) return $true$
    return $false$
endfunction

function ProveAll($Obs$)
    foreach ($[\varphi \to \psi, c_{\text{RU}}, c_{\text{KT}}] \in Obs$)
        if ($not$ Prove($\varphi \to \psi,\ c_{\text{RU}},\ c_{\text{KT}}$))
            return $false$;
    return $true$
endfunction
```

Figure 6.8: Top-Level DFS Proof Search Algorithm

they exceed the preset search bounds, so the proof procedure is incomplete, which is expected. Specifically, the algorithm consists of two cases: (Base Case) and (Recursive Call).

For (Base Case), procedure BasicProof is the exit point of the algorithm. For each proof goal, it firstly attempts a *basic proof*, i.e., to discharge by applying rule (PM) and then querying an SMT solver, where recursive symbols are treated as uninterpreted as in (SMT) proof rule. Intuitively, this step succeeds if the proof goal is simple enough such that a proof by matching can be achieved.

For (Recursive Call), when a basic proof fails, we collect all possible transformations of the proof goal, using (KT), (UNFOLD-R) rules, into a disjunction of conjunctions of sub-goals $OrSet$ (i.e., a set of goal sets)—here, we only present the least fixpoint reasoning. The current proof goal can be successfully discharged if there is one set $Obs \in OrSet$ whose goals can all be proved. The realization of the proof rules in our algorithm is straightforward, except for two noteworthy points:

1. (KT) applications will exhaustively search for all possible candidates.
2. When a proof goal has an unsatisfiable left-hand side, the proof goal is trivially true, which is denoted trivially_true in Figure 6.8, and is removed immediately.

Figure 6.8 essentially implements a (bounded) depth-first proof search, so the order in which the sets of goals $Obs \in OrSet$ are tried may affect performance greatly but not effectiveness, i.e. the ability to prove the proof goals of our proof framework. The algorithm is parametric in a procedure OrderByHeuristics (line 24) that controls the mentioned order. For the experiments considered in this paper, we use the following intuitive order, which follows the fact that our base case is reached by a successful basic proof.

We proceed by a number of passes. In each pass, we first order the goals within each $Obs \in OrderSet$. We then consider the order of $OrderSet$ by comparing the last goal in each set $Obs \in OrderSet$. Subsequent passes will not undo the work of the previous passes, but instead work on the goals and/or sets of goals which are tied in previous passes. Below are a few things to note.

1. Goals without recursive patterns on the right-hand side are prioritized.
2. Goals with recursive patterns on the right-hand side but not on the left-hand side are considered next.
3. Goals with fewer existential variables are prioritized.

6.3.2 Context matching algorithm

Procedure cm is used to check whether the inner context can be matched with the outer context. For example, suppose we have the following proof goal:

$$\vdash C_{outer}[\forall \tilde{y} \,.\, (C' \multimap \varphi')] \to \psi \tag{6.5}$$

$\mathsf{cm}(C_{outer}, C', \tilde{y})$ takes as inputs the outer context C_{outer}, the inner context C' and a list of quantifier variables $\tilde{y}$. To check if C' can be matched by (a part of) C_{outer}, it builds the following proof goal:

$$\vdash C_{outer} \to \exists \tilde{y} \,.\, C' \tag{6.6}$$

In (6.5), what we want is to initialize the universal variables $\forall \tilde{y}$ in order for the inner context C' to be matched with some part of C_{outer}. That is also the purpose of using the existential variables $\exists \tilde{y}$ in (6.6). The change of the quantifier $\tilde{y}$ from universal to existential is because we have moved the inner context C' from left-hand side to right-hand side.

To deal with (6.6), cm will call the modified version of the Prove function in Figure 6.8. The difference between the modified version and the Prove function is only on the returning result. Specifically, apart from returning *true* if the Prove function returns *true*, the modified version additionally (1) returns the remaining, unmatched part of the left-hand side , denoted C_{rest}, after consuming all the matched constraints from the right-hand side, and (2) collects the instantiation of $\tilde{y}$, denoted θ, when applying rule (PM). (Note that C_{rest} may contain structure patterns as we have seen in the SL examples.) Specifically, if we can prove (6.6), we have $\vdash C_{outer} \to C'\theta$. Furthermore, C_{rest} is the remaining part after removing the constraint of $C'\theta$ from C_{outer}, so we have $C_{rest}[C'\theta[C'\theta \multimap \varphi'\theta]] \to \psi$. As a result, we now can proceed to prove new proof goal $C_{rest}[\varphi'\theta] \to \psi$.

6.3.3 Pattern matching algorithm

Procedure pm, used by rule (PM), implements a naive, brute-force algorithm as shown in Figure 6.9 that does matching modulo associativity and/or associativity-and-commutativity. Procedure pm takes as input

- a list of "pattern" patterns $[\psi_i]_1^m \equiv [\psi_1, \ldots, \psi_m]$;
- a list of "term" patterns $[\varphi_j]_1^m \equiv [\varphi_1, \ldots, \varphi_m]$;
- a set of existential variables EV with $EV \cap \bigcup_1^n \mathit{freeVar}(\varphi_j) = \emptyset$ and $EV \subseteq \bigcup_1^m \mathit{freeVar}(\psi_i)$.

Then it returns Failure or the match result θ with $\mathsf{domains}(\theta) \subseteq EV$ and $\psi_i\theta \equiv \varphi_i$, for all i.

```
function $\mathtt{pm}([\psi_i]_1^m, [\varphi_i]_1^m, EV)$
  if $m = 0$ return $\{\,\}$
  if $\psi_1 \equiv \sigma(\tilde{\psi}_1)$ and $\varphi_1 \equiv \sigma'(\tilde{\varphi}_1)$
    if $\sigma \neq \sigma'$ return Failure
    else if $\mathsf{length}(\tilde{\psi}_1) \neq \mathsf{length}(\tilde{\varphi}_1)$
      return Failure
    else
      $[\psi_i']_1^{m'} = \tilde{\psi}_1 \cup [\psi_i]_1^m$
      $[\varphi_i']_1^{m'} = \tilde{\varphi}_1 \cup [\varphi_i]_1^m$
      return $\mathtt{pm}([\psi_i']_1^{m'}, [\varphi_i']_1^{m'}, EV)$
  if $\psi_1 \equiv x$ and $x \notin EV$
    if $\varphi_1 \equiv x$
      return $\mathtt{pm}([\psi_i]_2^m, [\varphi_i]_2^m, EV)$
    else
      return Failure
  if $\psi_1 \equiv x$ and $x \in EV$
    $[\psi_i']_2^m = [\psi_i]_2^m\{x \mapsto \varphi_1\}$
    $[\varphi_i']_2^m = [\varphi_i]_2^m\{x \mapsto \varphi_1\}$
    $\theta' = \mathtt{pm}([\psi_i']_2^m, [\varphi_i']_2^m, EV)$
    return $\{x \mapsto \varphi_1\} \cup \theta'$
  return Failure
endfunction
```

Figure 6.9: Pattern Matching Algorithm

6.4 EVALUATION

We implemented our proof framework in the 𝕂 framework (Section 2.15). 𝕂 has a modular notation for defining rewrite systems. Since our proof framework is essentially a rewriting system that rewrites/reduces proof goals to sub-goals, it is convenient to implement it in 𝕂.

We evaluated our prototype implementation using four representative logical systems for fixpoint reasoning: first-order logic extended with least fixpoints (LFP), separation logic (SL), linear temporal logic (LTL), and reachability logic (RL). Our evaluation plan is as follows. For SL, we used the 280 benchmark properties collected by the SL-COMP'19 competition [96]. These properties are entailment properties about various inductively-defined heap structures, including several hand-crafted, challenging structures. For LTL, we considered the (inductive) axioms in the complete LTL proof system (see, e.g., [100, 101]). For LFP and RL, we considered the program verification of a simple program `sum` that computes the total sum from 1 to a symbolic input n. We shall use two different encodings to capture the underlying transition relation: the LFP encoding defines it as a binary predicate and the RL encoding defines it as a reachability rule.

Before we discuss our evaluation results, we would like to point out that it would be unreasonable to expect that a unified proof framework can outperform the state-of-the-art provers and algorithms for all specialized domains from the first attempt. We believe that this is possible and within our reach in the near future, but it will likely take several years of sustained effort. We firmly believe that such effort will be worthwhile spending, because if successful then it will be transformative for the field of automated deduction and thus program verification. Here, we focus on demonstrating the generality of our proof framework. We shall also report the difficulties that we experienced.

Our first evaluation is based on the standard separation logic benchmark set collected by SL-COMP'19 [96]. These benchmarks are considered challenging because they are related to heap-allocated data structures along with user-defined recursive predicates crafted by participants to challenge the competitors. Among the benchmarks, we focus on the qf_shid_entl division that contains entailment problems about inductive definitions. This division is considered the hardest one, specifically because many of its tests require proofs by induction. As such, this division is a good case study for testing the generality of our generic proof framework. Furthermore, heap provers are currently considered to have the most powerful implementations of automated inductive reasoning, so we would not be far from the truth considering a comparison of our prototype with these as a comparison with the state-of-the-art in automated inductive reasoning.

To set up our prover for the SL benchmarks, we instantiate it with the set Γ^{SL} of axioms that captures SL, as given in Section 5.3. Note that the associativity and commutativity of $\varphi_1 * \varphi_2$ are handled by the built-in pattern matching algorithms (see Figure 6.9), so the most important axioms are the two specifying non-zero locations and no-overlapped heap unions. The experimental results show that our generic prover can prove 265 of the 280 benchmark tests, placing it third place among all participants.

Interestingly, we noted that (FRAME) is not necessary for most tests. Only 12 out of the 265 tests used (FRAME) reasoning. More experiments are needed to draw any firm conclusion, but it could be (FRAME) reasoning mostly improves performance, as it reduces the matching search space and thus proof search terminates faster, but does not necessarily increase the expressiveness of the prover. The 15 tests that our prover cannot handle come from the benchmarks of automata-based heap provers [89, 102]. These benchmarks demand more sophisticated SL-specific reasoning that require more complex properties about heaps/maps than what our prover can naively derive from the Γ^{SL} theory with its current degree of automation; while we certainly plan to handle those as well in the near future, we would like to note that they are not related to fixpoint reasoning, but rather to reasoning about maps. The two provers that outperform our generic prover, Songbird and S2S, are both specialized

Table 6.1: Selected separation logic properties, automatically proved by our prover

$\mathsf{sorted\_list}(x, min) \to \mathsf{list}(x)$ $\mathsf{sorted\_list}_1(x, len, min) \to \mathsf{list}_1(x, len)$ $\mathsf{sorted\_list}_1(x, len, min) \to \mathsf{sorted\_list}(x, min)$ $\mathsf{sorted\_ls}(x, y, min, max) * \mathsf{sorted\_list}(y, min_2) \land \ max \leq min_2 \ \to \ \mathsf{sorted\_list}(x, min)$
$\mathsf{lr}(x, y) * \mathsf{list}(y) \to \mathsf{list}(x)$ $\mathsf{lr}(x, y) \to \mathsf{ll}(x, y)$ $\mathsf{ll}(x, y) \to \mathsf{lr}(x, y)$ $\mathsf{ll}_1(x, y, len_1) * \mathsf{ll}_1(y, z, len_2) \to \ \mathsf{ll}_1(x, z, len_1{+}len_2)$ $\mathsf{lr}_1(x, y, len_1) * \mathsf{list}_1(y, len_2) \to \mathsf{list}_1(x, len_1{+}len_2)$ $\mathsf{ll}_1(x, last, len) * (last \mapsto new) \to \mathsf{ll}_1(x, new, len + 1)$
$\mathsf{dls}(x, y) * \mathsf{dlist}(y) \to \mathsf{dlist}(x)$ $\widehat{\mathsf{dls}}_1(x, y, len_1) * \widehat{\mathsf{dls}}_1(y, z, len_2) \to \widehat{\mathsf{dls}}_1(x, z, len_1{+}len_2)$ $\mathsf{dls}_1(x, y, len_1) * \mathsf{dlist}_1(y, len_2) \to \mathsf{dlist}_1(x, len_1{+}len_2)$
$\mathsf{avl}(x, hgt, min, max, balance) \to \mathsf{bstree}(x, hgt, min, max)$ $\mathsf{bstree}(x, height, min, max) \to \mathsf{bintree}(x, height)$

Table 6.2: Axioms in the complete LTL proof system, automatically proved by our prover

($\text{K}_\Box$)	$\Box(\varphi_1 \to \varphi_2) \to (\Box\varphi_1 \to \Box\varphi_2)$
(IND)	$\varphi \land \Box(\varphi \to \circ\varphi) \to \Box\varphi$
(U_1)	$\varphi_1 \, U \, \varphi_2 \to \diamond\varphi_2$
($\text{U}_2.1$)	$\varphi_1 \, U \, \varphi_2 \to \varphi_2 \lor \varphi_1 \land \circ(\varphi_1 \, U \, \varphi_2)$
($\text{U}_2.2$)	$\varphi_2 \lor \varphi_1 \land \circ(\varphi_1 \, U \, \varphi_2) \to \varphi_1 \, U \, \varphi_2$

for SL. Compared with generic provers such as CYCLIST [103], our prover proves 13 more tests.

Table 6.1 illustrates some of the more interesting SL properties that our prover can verify automatically. These are common lemmas about heap structures that arise and are collected when verifying real-world heap-manipulating programs. For example, the property on the first line says that a sorted list is also a list, a typical verification condition arising in formal verification. Table 6.1 also shows several proof goals about singly-linked lists and list segments (specified by predicates ls, list, ll, lr, etc.), doubly-linked lists and list segments (specified by predicates dls, dlist, etc.), and trees.

Our second study is to automatically prove the inductive axioms in the complete LTL proof system, shown in Table 6.2, whereas the proof tree of the most interesting of them, (IND), has been given in Section 6.2.4. Note that LTL is essentially a structure-less logic, as its formulas are only built from temporal operators and propositional connectives, and its models are infinite traces of states that have no internal structures and are modeled as "points". The structure-less-ness of LTL made fixpoint reasoning for it simpler, as no context reasoning or frame reasoning was needed.

Our final study considers a simple program `sum` that computes the total from 1 to a symbolic input n. We do the verification of `sum` following two approaches: RL and LFP. The reachability logic (RL) approach has been illustrated in Section 2.14. For LFP, program configurations are encoded as FOL terms and the program semantics is encoded as a binary FOL predicate that captures the transition relation. In particular, reachability is defined as a recursive predicate based on the semantics. Our prover then becomes a (language-independent) program verifier, different from Hoare-style verification (where a language-specific verification condition generator is required).

We ran these tests on a single core virtual machine with 8GB of RAM. The SL-COMP'19 tests took a total 13 hours to finish, including two outliers that took approximately one and three hours to complete. The two LTL tests took approximately three minutes, while the ten first order logic tests took seven minutes to complete and the sum program takes a minute to complete. To reiterate, we do not expect our prover to outperform specialized provers this early in its development. These results do, however, show that a unified, powerful and efficient proof framework is within reach.

Chapter 7: APPLICATIVE MATCHING μ-LOGIC (AML)

In an ideal unifying semantics-based language framework, all programming languages must have their formal syntax and semantics definitions. In addition, all execution and formal analysis tools of a given programming language L must be automatically generated from its formal definition Γ^L, where Γ^L is a logical theory that axiomatically define the static configurations and dynamic behaviors of all programs of L. As discussed in Section 2.15, the $\mathbb{K}$ framework is one of the many efforts in pursuing above vision of an ideal unifying semantics-based language framework. $\mathbb{K}$ has been used to define the complete formal semantics of many large programming languages and generate their execution and formal analysis tools.

A major research question has been this: *what is the right logical foundation of* $\mathbb{K}$*?* This is a challenging question to answer, for $\mathbb{K}$ is such a complex artifact whose implementation has over 500,000 lines of code in multiple programming languages. Previously, a tentative answer was given by matching logic (Section 2.13) and reachability logic (Section 2.14). Matching logic was used to specify and reason about the static configurations of programs as well as any (FOL) constraints over them. Reachability logic was used to specify and reason about the dynamic reachability properties. In particular, reachability logic has a language-independent proof system (Figure 2.12) that supports sound and relatively complete verification for all programming languages. Unfortunately, reachability logic cannot express more dynamic behaviors/properties such as liveness properties, which can be expressed using a temporal logic or modal μ-calculus. Besides the combination of matching logic and reachability logic, there are also other attempts to find a logical foundation for $\mathbb{K}$, including one using (double-pushout) graph transformations [104], one based on a translation to Isabelle [105], and one based on a translation to (coinduction in) Coq [106]. None of these are incorporated within $\mathbb{K}$'s codebase because none of them are satisfactory: not only they result in heavy translations with a big representational distance from the original definition, but also they focus on one aspect of $\mathbb{K}$ (e.g., reachability, or partial correctness, or coinductive).

To overcome these limitations, we propose matching μ-logic in Chapter 4 that not only unifies matching logic and reachability logic but also captures many important logics and calculi as its theories. We carry out an extensive study on its expressive power and show that LFP, separation logic with recursive predicates, modal μ-calculus, temporal logics, dynamic logic, λ-calculus, and type systems can be defined in matching μ-logic as theories. Therefore, matching μ-logic is a good candidate for serving as the logical foundation of $\mathbb{K}$.

However, matching μ-logic suffers from at least two main technical inconveniences. Firstly, matching μ-logic is more complex than necessary. As a many-sorted logic, matching μ-logic has theories that can contain multiple, sometimes infinitely many sorts, each with its own carrier set in models. Matching μ-logic uses many-sorted symbols, such as $\sigma \in \Sigma_{s_1 \ldots s_n, s}$, of fixed arities, to build patterns of the appropriate sorts. This places a burden on implementations, which need to store the sorts and the symbols arities, carry out well-formedness checking, and implement a more-complex-than-needed proof checker. More importantly, the fact that the structure of a many-sorted universe is hardwired in matching μ-logic actually makes it less general, when it comes to more complex sort structures such as parametric sorts or ordered sorts (i.e., subsorts). As an example, suppose we have a sort Nat of natural numbers and we want to define parametric lists. To do that, we have to introduce a new sort $\mathsf{List}\{s\}$ for every sort s, where $\mathsf{List}\{s\}$ is the sort of all the lists over elements of sort s. As a result, we introduce infinitely many sorts: Nat, $\mathsf{List}\{\mathsf{Nat}\}$, $\mathsf{List}\{\mathsf{List}\{\mathsf{Nat}\}\}$, etc.. Even though we can handle infinitely many sorts in theory, no implementation can handle them unless we come up with certain finite representations. In addition, we have to introduce infinitely many symbols for the common parametric operations over the parametric lists and define infinite many axioms for them (we only show the axioms for append as an example):

$$\begin{aligned}
&\mathsf{nil}\{s\} \in \Sigma_{\epsilon,\mathsf{List}\{s\}} \\
&\mathsf{cons}\{s\} \in \Sigma_{s\,\mathsf{List}\{s\},\mathsf{List}\{s\}} \\
&\mathsf{append}\{s\} \in \Sigma_{\mathsf{List}\{s\}\,\mathsf{List}\{s\},\mathsf{List}\{s\}} \\
&\mathsf{append}\{s\}(\mathsf{cons}\{s\}(x : s, l : \mathsf{List}\{s\}), l' : \mathsf{List}\{s\}) \\
&\quad = \mathsf{cons}\{s\}(x : s, \mathsf{append}\{s\}(l : \mathsf{List}\{s\}, l' : \mathsf{List}\{s\}))
\end{aligned}$$

This is at best inconvenient for matching μ-logic implementations. Either we incorporate parametric lists as a built-in feature into the implementations, or we invent some ad-hoc meta-level notation to specify the infinite theory of parametric lists in some finite way. Neither approach is optimal: the former lacks generality while the latter is heavy and superficial. Most many-sorted FOL systems forgo parametricity all together.

Secondly, matching μ-logic enforces a strict separation among elements, sorts, and symbols. Intuitively, elements represent data; sorts represent the types of data; and symbols represent operations or predicates over data. Thus in matching μ-logic, there is a distinction among data, types, and operations/predicates. While such a distinction is beneficial in a classic, algebraic setting, it can get inconvenient when it comes to, say, functional programming, where functions are also first-class citizens as other types of data.

The purpose of this chapter is to introduce a methodological solution, called *applicative matching μ-logic*, abbreviated as AML, which addresses the above technical inconveniences.

We call AML a methodological solution because it is not an extension nor a modified version of matching μ-logic. Instead, AML is an instance of matching μ-logic where we restrict the use of sorts and many-sorted symbols to an absolute minimum. In AML, we define only one sort for all patterns and enforce all symbols to be constant symbols except for one, which is a binary symbol called *application* (see Section 7.1). We will show that AML has the same expressive power as matching μ-logic despite it being much more simpler. What are hardwired in matching μ-logic, such as sorts and many-sorted symbols, can now be axiomatically defined in AML as theories, just like how other mathematical instruments such as equality and functions are axiomatically defined in matching μ-logic as theories. AML is matching μ-logic at extreme simplicity without loss of expressive power.

It should be noted that we do not intend to argue which is better, AML or matching μ-logic. As said, AML is not a new logic, but rather a restricted use of matching μ-logic, a self-restraint version of it. This is why we call AML a *methodology.* Recall the above parametric lists example. Now we have two ways to define them. One is to use the full power of matching μ-logic by introducing infinitely many sorts and symbols like we have seen earlier. The other is to use AML and axiomatically define sorts and the many-sorted symbols. Both approaches have their merits. The former reduces the handling of sorts to the logic while the latter can give us a finite theory, which is easier to handle in implementations. Another example is order-sorted structures, where a sort s can be a *subsort* of another sort s', written $s \leq s'$. It is enforced that the carrier set of s is a subset of that of s'. Here, the many-sorted infrastructure of matching μ-logic has few advantages but burdens. To define subsorts we need to introduce a function $c_s^{s'} \colon s \to s'$, called a coercion function from s to s', for every $s \leq s'$. Furthermore, we need to specify that $c_s^{s'}$ is injective. For any $s \leq s' \leq s''$, we also need to specify the following triangle property $\mathsf{inj}_{s'}^{s''}(\mathsf{inj}_s^{s'}(x : s)) = \mathsf{inj}_s^{s''}(x : s)$. All these extra mechanisms regarding the coercion functions will not be needed in AML.

In what follows we will present AML in full detail. We will present a reduction from matching μ-logic to AML, which implies that AML has the same expressive power as matching μ-logic. Then we will revisit the examples of subsorts and parametric sorts and use them to illustrate the unique advantage of AML (as a methodology) when it comes to specifying more advanced sort structures. Finally, we will present a proof checker based on AML. Thanks to the simplicity of AML, our proof checker has only 240 lines of code in Metamath [107].

7.1 AML AS AN INSTANCE OF MATCHING μ-LOGIC

AML is an instance of matching μ-logic when the signature (S, Σ) has the form $S = \{\star\}$ and $\Sigma = C \cup \{\mathsf{app}\}$, where $\star$ is a (dummy) sort, C is a set of constant symbols, and $\mathsf{app} \in \Sigma_{\star\star,\star}$

is a binary symbol, called *application*. The syntax, semantics, and proof system of AML follows from the syntax, semantics, and proof system of matching μ-logic over the above restricted signatures, but we can introduce more simplified notations for AML.

Firstly, we can drop the dummy sort $\star$ and keep things unsorted. In particular, we have a set EV of unsorted element variables and a set SV of unsorted set variables. We denote them by x, X, etc., instead of $x{:}\star$ or $X:\star$. Secondly, app is the only non-constant symbol in AML so let us add it directly to the syntax of AML patterns. Furthermore, we write $\varphi_1\,\varphi_2$ for $\mathsf{app}(\varphi_1, \varphi_2)$ and write $\varphi_1\,\varphi_2\,\ldots\,\varphi_n$ for $(\ldots(\varphi_1\,\varphi_2)\,\ldots\,\varphi_n)$, i.e., application is left associative. Now we can present AML as follows:

Definition 7.1. An AML signature Σ is a set of constant symbols. The set of AML Σ-patterns, written AMLPATTERN(Σ), 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 $\mu X\,.\,\varphi$ requires that φ is positive in X. An AML Σ-model is a tuple $(M, \{\_\bullet_M\_\}, \{\sigma_M \mid \sigma \in \Sigma\})$ where M is a nonempty set, $\_\bullet_M\_ : M \times M \to \mathcal{P}(M)$, and $\sigma_M \subseteq M$ for every $\sigma \in \Sigma$. An AML valuation $\rho = (\rho_{EV}, \rho_{SV})$ where $\rho_{EV} : EV \to M$ and $\rho_{SV} : SV \to \mathcal{P}(M)$. Given M and ρ, the evaluation of $\varphi \in$ AMLPATTERN(Σ) is also denoted by $|\varphi|_{M,\rho}$ and is defined the same way in matching μ-logic.

The name of AML comes from applicative structures, which are algebras with a binary operation for application. Applicative structures are important in the study of combinations and λ-calculus. Here, we only show that applicative structures are an special case of AML models, where the application symbol is interpreted as a total function.

Proposition 7.1. *An applicative structure $(A, \_\bullet_A\_)$ is a pair of a nonempty set A and a function $\_\bullet_A\_ : A \times A \to A$ [29, Definition 5.1.1]. Then, applicative structures are AML $\emptyset$-models M where $\mathsf{card}(\mathsf{app}_M(a, b)) = 1$ for all $a, b \in M$.*

The common mathematical instruments such as sorts, equality, membership, and functions can be defined in AML as theories. Since we have shown that all of them can be defined in matching μ-logic as theories, we only need to show that matching μ-logic can be defined in AML as theories. We do that in Section 7.2.

7.2 DEFINING MATCHING μ-LOGIC IN AML

To define matching μ-logic in AML, we need to define sorts and many-sorted symbols. The idea is straightforward. We first introduce a distinguished symbol $\top\_$ called the inhabitant symbol, and write $((\top\_)\,\varphi)$, which is the result of applying $\top\_$ to φ using the AML application symbol, as $\top_\varphi$. Then for every sort s we introduce it as a (constant) symbol in AML. Then the pattern $\top_s$ is matched by all the elements of sort s.

Now we define in detail the reduction from matching μ-logic to AML. Let us fix a matching μ-logic signature (S^{MmL}, Σ^{MmL}) where $EV^{MmL} = \{EV_s^{MmL}\}_{s \in S^{MmL}}$ and $SV^{MmL} = \{SV_s^{MmL}\}_{s \in S^{MmL}}$ are the two S^{MmL}-indexed sets of element and set variables, respectively. Let $EV^{AML} = \{x^s \mid x : s \in EV_s^{MmL}\}$ be the set of AML unsorted element variables. Similarly, let $SV^{AML} = \{X^s \mid X : s \in SV_s^{MmL}\}$ be the set of AML unsorted set variables. We keep track of the original sorts as superscripts so we can restore the sort information after translation. We also feel free to use x, y, X, Y, etc. in AML whenever we are defining AML axioms. Let $\Sigma^{AML} = \{\lceil\_\rceil, \top\_\} \cup S^{MmL} \cup \Sigma^{MmL}$, where $\lceil\_\rceil$ and $\top\_$ are two distinguished symbols. Let Γ^{AML} be the AML theory that includes the following axioms:

(Definedness, AML version) $\lceil x \rceil$

(Nonempty Carrier Set) $\top_s \neq \emptyset$ for each $s \in S^{MmL}$

(Symbol Arity) $\sigma\, \top_{s_1} \ldots \top_{s_n} \subseteq \top_s$ for each $\sigma \in \Sigma^{MmL}_{s_1 \ldots s_n, s}$

We define the translation from matching μ-logic (S^{MmL}, Σ^{MmL})-patterns to AML (Σ^{AML})-patterns as follows:

$$\begin{aligned}
AML(\varphi_s) &= wellsorted(\varphi_s) \to (AML_2(\varphi_s) = \top_s) \\
wellsorted(\varphi_s) &= \bigwedge_{x : s' \in EV^{MmL}} x^{s'} \in \top_s \wedge \bigwedge_{X : s' \in SV^{MmL}} X^{s'} \subseteq \top_s \\
AML_2(x : s) &= x^s \\
AML_2(X : s) &= X^s \\
AML_2(\sigma(\varphi_1, \ldots, \varphi_n)) &= \sigma\, AML_2(\varphi_1) \ldots AML_2(\varphi_n) \\
AML_2(\varphi_s \wedge \varphi'_s) &= AML_2(\varphi_s) \wedge AML_2(\varphi'_s) \\
AML_2(\neg \varphi_s) &= \neg_s AML_2(\varphi_s) \equiv \neg AML_2(\varphi_s) \wedge \top_s \\
AML_2(\exists x : s' \,.\, \varphi) &= \exists x^s \,.\, (x^s \in \top_s) \wedge AML_2(\varphi) \\
AML_2(\mu X : s \,.\, \varphi) &= \mu X^s \,.\, AML_2(\varphi) \\
AML(\Gamma) &= \{AML(\psi) \mid \psi \in \Gamma\}
\end{aligned}$$

Proposition 7.2. *For any matching μ-logic theory Γ and pattern φ, $\Gamma \vDash \varphi$ iff $\Gamma^{AML} \cup AML(\Gamma) \vDash AML(\varphi)$.*

Proof. We first define a translation from AML $(\Sigma^{AML}, \Gamma^{AML})$-models to matching μ-logic $(\Sigma^{MmL}, \Sigma^{MmL})$-models. Consider an arbitrary $(\Sigma^{AML}, \Gamma^{AML})$-model

$$M^{AML} = (M^{AML}, \{\_\bullet_{M^{AML}}\_\}, \{\sigma_{M^{AML}}\}_{\sigma \in \Sigma^{AML}})$$

We define the corresponding $(\Sigma^{MmL}, \Sigma^{MmL})$-model

$$M^{MmL} = (\{M^{MmL}_s\}_{s \in S^{MmL}}, \{\sigma_{M^{MmL}}\}_{\sigma \in \Sigma^{MmL}})$$

where

1. $M^{MmL}_s = |\top_s|_{M^{AML}}$ for every $s \in S^{MmL}$;
2. $\sigma_{M^{MmL}}(a_{s_1}, \ldots, a_{s_n}) = |\sigma\, x_1\, \ldots\, x_n|_{M^{AML}, \rho}$ where ρ is any valuation such that $\rho(x_i) = a_{s_i}$, for all $a_{s_i} \in M^{MmL}_{s_i}$, $1 \leq i \leq n$.

The above definition is well-defined because of (Nonempty Carrier Set) and (Symbol Arity). Also note that the above translation is surjective, i.e., for any $(\Sigma^{MmL}, \Sigma^{MmL})$-model M, we can find a $(\Sigma^{AML}, \Gamma^{AML})$-model M^{AML}, whose corresponding $(\Sigma^{MmL}, \Sigma^{MmL})$-model $M^{MmL} = M$. Furthermore, for any M^{MmL}-valuation $\rho^{MmL} = (\rho^{MmL}_{EV}, \rho^{MmL}_{SV})$, we define the corresponding M^{AML}-valuation $\rho^{\mathrm{AML}} = (\rho^{\mathrm{AML}}_{EV}, \rho^{\mathrm{AML}}_{SV})$ by letting $\rho^{\mathrm{AML}}_{EV}(x^s) = \rho^{MmL}_{EV}(x : s)$ and $\rho^{\mathrm{AML}}_{SV}(X^s) = \rho^{MmL}_{SV}(X : s)$.

Next, we show that for any matching μ-logic formula φ and M^{MmL}-valuation $\rho^{MmL} = (\rho^{MmL}_{EV}, \rho^{MmL}_{SV})$,

$$|\varphi|_{M^{MmL}, \rho^{MmL}} = |AML_2(\varphi)|_{M^{AML}, \rho^{\mathrm{AML}}}$$

by structural induction on φ. If φ is $x : s$ or $X : s$, the conclusion holds by the definition of ρ^{AML}. If φ is $\sigma(\varphi_1, \ldots, \varphi_n)$, the conclusion holds by the definition of $\sigma_{M^{MmL}}$. If φ is $\varphi_1 \wedge \varphi_2$ or $\neg\varphi_1$, the conclusion holds by the semantics of matching μ-logic and AML; note that it is important that the AML translation for $\neg$ uses $\neg_s$, which restricted the result within sort s. If φ is $\exists x : s' \,.\, \varphi$, the conclusion holds by the semantics of matching μ-logic and AML, as well as the definition of $M^{MmL}_{s'}$. If φ is $\mu X : s \,.\, \varphi$, the conclusion holds by noting that the functions $\mathcal{F}^{AML} \colon \mathcal{P}(M^{AML}) \to \mathcal{P}(M^{AML})$ and $\mathcal{F}^{AML} \colon \mathcal{P}(M^{MmL}_s) \to \mathcal{P}(M^{MmL}_s)$ given by

$$\begin{aligned} \mathcal{F}^{MmL}(A) &= |\varphi|_{M^{MmL}, \rho[A/X : s]} && \text{for } A \subseteq M^{AML} \\ \mathcal{F}^{AML}(A_s) &= |AML_2(\varphi)|_{M^{AML}, \rho[A/X^s]} && \text{for } A_s \subseteq M^{MmL}_s \end{aligned}$$

are equal over M^{MmL}_s, i.e., $\mathcal{F}^{AML}|_{M^{MmL}_s} = \mathcal{F}^{MmL}$; this is proved by applying the induction hypothesis. Furthermore, $\mathbf{lfp}\, \mathcal{F}^{MmL} = \mathbf{lfp}\, \mathcal{F}^{AML}$. This is because $\mathcal{F}^{MmL}$ is essentially a restriction of $\mathcal{F}^{AML}$ to a smaller domain, i.e., $M_s^{MmL} \subseteq M^{AML}$, so $\mathbf{lfp}\,\mathcal{F}^{AML} \subseteq \mathbf{lfp}\,\mathcal{F}^{MmL}$. However, it cannot be the case that $\mathbf{lfp}\,\mathcal{F}^{AML} \subsetneq \mathbf{lfp}\,\mathcal{F}^{MmL}$, because $\mathbf{lfp}\,\mathcal{F}^{AML}$, which we know is include by $\mathbf{lfp}\,\mathcal{F}^{MmL}$, which is then included by M_s^{MmL}, must also be a fixpoint of $\mathcal{F}^{MmL}$. Since $\mathbf{lfp}\,\mathcal{F}^{MmL}$ is the least fixpoint, we conclude that $\mathbf{lfp}\,\mathcal{F}^{MmL} = \mathbf{lfp}\,\mathcal{F}^{AML}$. And thus, we finish the structural induction.

Next, we show that for any matching μ-logic formula φ, $M^{MmL} \vDash \varphi$ iff $M^{AML} \vDash AML(\varphi)$. This is proved by noting that for any M^{MmL}-valuation ρ^{MmL} and its corresponding M^{AML}-valuation ρ^{AML}, $|\mathsf{ws}(\varphi)|_{M^{AML},\rho^{\mathrm{AML}}} = M^{AML}$. Furthermore, for an arbitrary M^{AML}-valuation ρ such that $|\mathsf{ws}(\varphi)|_{M^{AML},\rho} = M^{AML}$, we can find an M^{MmL}-valuation ρ^{MmL} whose corresponding M^{AML}-valuation $\rho^{\mathrm{AML}} \overset{\mathit{freeVar}(\varphi)}{\sim} \rho$.

Finally, we conclude that $\Gamma \vDash \varphi$ iff $\Gamma^{AML} \cup AML(\Gamma) \vDash AML(\varphi)$, by noting that the translation from M^{AML} to M^{MmL} is surjective. QED.

7.3 CASE STUDY: DEFINING ADVANCED SORT STRUCTURES IN AML

We have seen how many-sorted structures can be defined as AML theories. In this section we show how to define more advanced sort/type structures as AML theories, including subsorts, parametric sorts, function types, and dependent types.

7.3.1 Defining subsorts

For sorts s and s', we say that s is a *subsort* of s', written $s \le s'$, if $M_s \subseteq M_{s'}$. Since in AML, the carrier sets of s and s' are expressed by $\top_s$ and $\top_{s'}$, respectively, it is straightforward to define the subsort relation $s \le s'$ by

$$(\textsc{Subsort}) \qquad \top_s \subseteq \top_{s'}$$

More interestingly, we can axiomatically define *subsort overloading* of operations. For example, let Nat and Int be two sorts with the axiom $\top_{\mathsf{Nat}} \subseteq \top_{\mathsf{Int}}$ stating that Nat is a subsort of Int. We can define plus as an overloaded operation over Nat and Int as follows:

$$\begin{aligned} &(\textsc{Plus, Arity } 1) &&\mathsf{plus}\,\top_{\mathsf{Nat}}\,\top_{\mathsf{Nat}} \subseteq \top_{\mathsf{Nat}} \\ &(\textsc{Plus, Arity } 2) &&\mathsf{plus}\,\top_{\mathsf{Int}}\,\top_{\mathsf{Int}} \subseteq \top_{\mathsf{Int}} \end{aligned}$$

Using the above axioms, we can prove that $\mathsf{plus}(1,2)$ has sort both Nat and Int while $\mathsf{plus}(-1,-2)$ has only sort Int but not Nat.

It is known (see, e.g., [108]) that ordered sorts can be defined in a many-sorted setting, where the subsort relation is captured by the *coercion functions* $c_s^{s'} \in \Sigma_{s,s'}$ for all $s \leq s'$. Intuitively, $c_s^{s'}$ denotes the embedding from sort s to sort s'. This approach, however, is not practically useful, as noticed in [109, pp. 9]. For example, consider three sorts $s \leq s' \leq s''$, a constant a of sort s, and a function $f \in \Sigma_{s'',s''}$. Then, the term $f(x)$ has multiple parses when translated to FOL, e.g.: $f(c_s^{s''}(a))$ and $f(c_{s'}^{s''}(c_s^{s'}(a)))$. This means that all tools for OSA based on FOL need to do reasoning modulo the triangle property $c_s^{s''}(a) = c_{s'}^{s''}(c_s^{s'}(a))$, which is inconvenient and causes huge overhead. In contrast, AML provides a more succinct and native approach to handling subsorts, by directly defining subsort axioms, without needing to introduce coercion functions.

7.3.2 Defining parametric sorts

A parametric sort, such as $\mathsf{List}\{s\}$, can be viewed as a function over sorts. Indeed, given a sort s, $\mathsf{List}\{s\}$ returns its list sort. Since AML treats sorts and functions as regular elements, parametric sorts can be directly defined as functions. Let us define an AML symbol Sort whose (intended) elements are sorts. We add an axiom $\mathsf{Nat} \in \mathsf{Sort}$ so Nat becomes a sort. Then, we define a function $\mathsf{List}\colon \mathsf{Sort} \to \mathsf{Sort}$, called *sort constructor*, which takes a sort s and produces the sort $\mathsf{List}\, s$ of lists parametric in s. Standard list operations can be also defined as functions:

$$\forall s : \mathsf{Sort} \,.\, \exists l' : \mathsf{List}\, s \,.\, \mathsf{nil} = l'$$
$$\forall s : \mathsf{Sort} \,.\, \forall x : s \,.\, \forall l : \mathsf{List}\, s \,.\, \exists l' : \mathsf{List}\, s \,.\, \mathsf{cons}\, x\, l = l'$$
$$\forall s : \mathsf{Sort} \,.\, \forall l_1 : \mathsf{List}\, s \,.\, \forall l_2 : \mathsf{List}\, s \,.\, \exists l' : \mathsf{List}\, s \,.\, \mathsf{append}\, l_1\, l_2 = l'$$

Note that the above axioms are similar to the (Function) axioms in matching μ-logic but here s is a variable ranging over Sort.

7.3.3 Defining function types

Functions are also elements in AML, and function sorts can be built by $\mathsf{Function}\colon \mathsf{Sort} \times \mathsf{Sort} \to \mathsf{Sort}$, with the following axiom

$$(\textsc{Function Sort}) \quad \forall s : \mathsf{Sort} \,.\, \forall s' : \mathsf{Sort} \,.\, \top_{\mathsf{Function}\, s\, s'} = \exists f \,.\, f \wedge \forall x : s \,.\, \exists y : s' \,.\, f x = y$$

stating that $\mathsf{Function}\, s\, s'$ consists of all f that behaviors as a function from s to s'. As an example, we define two higher-order list operations: *fold* and *map*, which are common in functional programming languages.

$$\forall s : \mathsf{Sort} \,.\, \forall s' : \mathsf{Sort} \,.\, \mathsf{fold} \colon \mathsf{Function}\, s'\, s\, s' \times s' \times \mathsf{List}\, s \to s'$$

$$\forall f : \mathsf{Function}\, s'\, s\, s' \,.\, \forall x : s' \,.\, \mathsf{fold}\, f\, x\, \mathsf{nil} = x$$

$$\forall g : \mathsf{Function}\, s\, s' \,.\, \forall y : s \,.\, \forall l : \mathsf{List}\, s \,.\, \mathsf{fold}\, f\, x\, (\mathsf{cons}\, y\, l) = \mathsf{fold}\, f\, (fxy)\, l$$

$$\forall s : \mathsf{Sort} \,.\, \forall s' : \mathsf{Sort} \,.\, \mathsf{map} \colon \mathsf{Function}\, s\, s' \times \mathsf{List}\, s \to \mathsf{List}\, s'$$

$$\forall g : \mathsf{Function}\, s\, s' \,.\, \mathsf{map}\, g\, \mathsf{nil} = \mathsf{nil}$$

$$\forall f : \mathsf{Function}\, s'\, s\, s' \,.\, \forall x : s' \,.\, \forall y : s \,.\, \forall l : \mathsf{List}\, s \,.\, \mathsf{map}\, g\, (\mathsf{cons}\, y\, l) = \mathsf{cons}\, (g\, y)\, (\mathsf{map}\, g\, l)$$

7.3.4 Defining dependent types

Dependent types are also functions over sorts/types except that the parameters are data instead of sorts. That, however, makes no big difference in AML, for it makes no distinction between elements, sorts, and operations at all. All of them are are uniformly defined using patterns. Therefore, we can define dependent types the same way we define parametric sorts. As an example, suppose we want to define a dependent sort MInt of *machine integers*, such that $\mathsf{MInt}\, n$ for $n \geq 1$ is the sort of machine integers of size n, i.e., natural numbers less than 2^n. For clarity, we define a new sort Size for positive natural numbers and axiomatize MInt as follows:

$$\top_{\mathsf{Size}} = \mathsf{succ}\, \top_{\mathsf{Nat}}$$

$$\mathsf{MInt} \colon \mathsf{Size} \to \mathsf{Sort}$$

$$\forall n : \mathsf{Size} \,.\, \top_{\mathsf{MInt}\, n} = \exists x : \mathsf{Nat} \,.\, x \wedge x < \mathsf{power2}\, n$$

where $\mathsf{power2} \colon \mathsf{Nat} \to \mathsf{Nat}$ (power of 2) and _<_ (less-than) are defined in the usual way. We can then define functions over machine integers, such as mplus and mmult, by defining their arities and then *reusing* the addition plus and the multiplication mult over natural numbers:

$$\forall n : \mathsf{Size} \,.\, \mathsf{mplus} \colon \mathsf{MInt}\, n \times \mathsf{MInt}\, n \to \mathsf{MInt}\, (\mathsf{succ}\, n)$$

$$\forall n : \mathsf{Size} \,.\, \forall x : \mathsf{MInt}\, n\, \forall y : \mathsf{MInt}\, n \,.\, \mathsf{mplus}\, x\, y = \mathsf{plus}\, x\, y$$

$$\forall n : \mathsf{Size} \,.\, \forall m : \mathsf{Size} \,.\, \mathsf{mmult} \colon \mathsf{MInt}\, n \times \mathsf{MInt}\, m \to \mathsf{MInt}\, (\mathsf{plus}\, n\, m)$$

$$\forall n : \mathsf{Size} \,.\, \forall m : \mathsf{Size} \,.\, \forall m : \mathsf{Size}\, \forall x : \mathsf{MInt}\, n\, \forall y : \mathsf{MInt}\, m \,.\, \mathsf{mmult}\, x\, y = \mathsf{mult}\, x\, y$$

```
$c \imp ( ) #Pattern |- $.

$v ph1 ph2 ph3 $.
ph1-is-pattern $f #Pattern ph1 $.
ph2-is-pattern $f #Pattern ph2 $.
ph3-is-pattern $f #Pattern ph3 $.
imp-is-pattern
  $a #Pattern ( \imp ph1 ph2 ) $.

axiom-1
  $a |- ( \imp ph1 ( \imp ph2 ph1 ) ) $.

axiom-2
  $a |- ( \imp ( \imp ph1 ( \imp ph2 ph3 ) )
        ( \imp ( \imp ph1 ph2 )
               ( \imp ph1 ph3 ) ) ) $.

${
  rule-mp.0 $e |- ( \imp ph1 ph2 ) $.
  rule-mp.1 $e |- ph1 $.
  rule-mp   $a |- ph2 $.
$}
imp-refl $p |- ( \imp ph1 ph1 )
$=
  ph1-is-pattern ph1-is-pattern
  ph1-is-pattern imp-is-pattern
  imp-is-pattern ph1-is-pattern
  ph1-is-pattern imp-is-pattern
  ph1-is-pattern ph1-is-pattern
  ph1-is-pattern imp-is-pattern
  ph1-is-pattern imp-is-pattern
  imp-is-pattern ph1-is-pattern
  ph1-is-pattern ph1-is-pattern
  imp-is-pattern imp-is-pattern
  ph1-is-pattern ph1-is-pattern
  imp-is-pattern imp-is-pattern
  ph1-is-pattern ph1-is-pattern
  ph1-is-pattern imp-is-pattern
  ph1-is-pattern axiom-2
  ph1-is-pattern ph1-is-pattern
  ph1-is-pattern imp-is-pattern
  axiom-1 rule-mp ph1-is-pattern
  ph1-is-pattern axiom-1 rule-mp
$.
```

Figure 7.1: Metamath Formalization of AML (Extract)

7.4 AML PROOF CHECKER

We present an AML proof checker implemented in Metamath [107]. Metamath is a tiny language to state abstract mathematics and their proofs in a machine-checkable style. We use Metamath to formalize the syntax and proof system of AML and encode AML proofs. Metamath is known for its simplicity and efficient proof checking. Metamath proof checkers can be implemented in a few hundreds lines of code and can check thousands of theorems in a second. Our formalization follows closely the syntax of AML. We also need to formalize some metalevel operations such as free variables and capture-avoiding substitution. An innovative contribution is a generic way to handling notations.

7.4.1 Metamath overview

At a high level, a Metamath source file consists of a list of *statements*. The main ones are:

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

Figure 7.1 defines the fragment of AML with only implications. We declare five constants in a row in line 1, where `\imp`, `(`, and `)` build the syntax, `#Pattern` is the type of patterns, and `|-` is the provability relation. We declare three metavariables of patterns in lines 3-6, and the syntax of implication $\varphi_1 \to \varphi_2$ as `( \imp ph1 ph2 )` in line 7. Then, we define AML proof rules as Metamath axioms. For example, lines 18-22 define the rule (Modus Ponens). In line 23, we show an example (meta-)theorem and its formal proof in Metamath. The theorem states that $\vdash \varphi_1 \to \varphi_1$ holds, and its proof (lines 25-43) is a sequence of labels referring to the previous axiomatic/provable statements.

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

As an example, we look at the first 5 labels of the proof in Figure 7.1, line 25:

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

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

7.4.2 Main definitions

We now go through the main definitions of AML in Metamath and emphasize some highlights. The entire formalization has 200 lines of Metamath code, as shown in Section 7.4.3.

The syntax of AML patterns is formalized below:

```
$c \bot \imp \app \exists \mu ( ) $.
var-is-pattern       $a #Pattern xX $.
symbol-is-pattern    $a #Pattern sg0 $.
```

```
bot-is-pattern      $a #Pattern \bot $.
imp-is-pattern      $a #Pattern ( \imp ph0 ph1 ) $.
app-is-pattern      $a #Pattern ( \app ph0 ph1 ) $.
exists-is-pattern   $a #Pattern ( \exists x ph0 ) $.
${  mu-is-pattern.0 $e #Positive X ph0 $.
    mu-is-pattern   $a #Pattern ( \mu X ph0 ) $.  $}
```

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

We need the following metalevel operations and/or assertions: (1) positive (and negative) occurrences of variables; (2) free variables; (3) capture-avoiding substitution; (4) application contexts; (5) notations. Item 1 is needed to define the syntax of $\mu X \,.\, \varphi$, while Items 2-5 are needed to define the proof system. As an example, we show how to define capture-avoiding substitution. We first define a Metamath constant

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

which serves as an assertion symbol. The intuition of #Substitution is that if we can prove #Substitution ph ph' ph'' xX, then we have ph ≡ ph'[ph''/xX]. The definition is given based on the structure of ph'. For example, the following defines #Substitution when ph' is an implication:

```
${ substitution-imp.0  $e #Substitution ph1 ph3 ph0 xX $.
   substitution-imp.1  $e #Substitution ph2 ph4 ph0 xX $.
   substitution-imp
     $a #Substitution ( \imp ph1 ph2 ) ( \imp ph3 ph4 ) ph0 xX $. $}
```

When ph' is $\exists x \,.\, \varphi$ or $\mu X \,.\, \varphi$, we need to consider α-renaming to avoid variable capture. We show the case when ph' is $\exists x \,.\, \varphi$ below:

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

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

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

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

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

where we declare the constant `\not`, add it to the pattern syntax, and then define the sugaring/desugaring equivalence $\neg\varphi \equiv \varphi \to \bot$. We define all notations as above using `#Notation`.

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

```
notation-reflexivity $a #Notation ph0 ph0 $.
${ notation-symmetry.0 $e #Notation ph0 ph1 $.
   notation-symmetry   $a #Notation ph1 ph0 $. $}
${ notation-transitivity.0 $e #Notation ph0 ph1 $.
   notation-transitivity.1 $e #Notation ph1 ph2 $.
   notation-transitivity   $a #Notation ph0 ph2 $. $}
```

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

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

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

With metalevel assertions and notations, it is now straightforward to formalize the AML proof rules. We have seen the formalization of (MODUS PONENS) in Figure 7.1. In the following, we formalize the fixpoint proof rule (KANASTER-TARSKI), whose premises use capture-avoiding substitution:

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

Note that these proof rules collectively define the provability predicate `|-`. We also add the following axiom so that `#Notation` also preserves provability:

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

7.4.3 Entire source code

We present the entire 200-line Metamath formalization of AML.

```
$( MATCHING LOGIC PROOF CHECKER has 200 LOC $)
$c #Pattern #ElementVariable #SetVariable #Variable #Symbol $.
$v ph0 ph1 ph2 ph3 ph4 ph5 x y X Y xX yY sg0 $.
ph0-is-pattern $f #Pattern ph0 $. ph1-is-pattern $f #Pattern ph1 $.
ph2-is-pattern $f #Pattern ph2 $. ph3-is-pattern $f #Pattern ph3 $.
ph4-is-pattern $f #Pattern ph4 $. ph5-is-pattern $f #Pattern ph5 $.
x-is-element-var $f #ElementVariable x $.
y-is-element-var $f #ElementVariable y $.
X-is-element-var $f #SetVariable X $. Y-is-element-var $f #SetVariable Y $.
xX-is-var $f #Variable xX $. yY-is-var $f #Variable yY $.
sg0-is-symbol $f #Symbol sg0 $.
element-var-is-var $a #Variable x $. set-var-is-var $a #Variable X $.
var-is-pattern     $a #Pattern xX $. symbol-is-pattern  $a #Pattern sg0 $.
$c #Positive #Negative #Fresh #ApplicationContext #Substitution #Notation |- $.
$c \bot \imp \app \exists \mu ( ) $. bot-is-pattern $a #Pattern \bot $.
imp-is-pattern $a #Pattern ( \imp ph0 ph1 ) $.
app-is-pattern $a #Pattern ( \app ph0 ph1 ) $.
exists-is-pattern $a #Pattern ( \exists x ph0 ) $.
${ mu-is-pattern.0 $e #Positive X ph0 $.
   mu-is-pattern   $a #Pattern ( \mu X ph0 ) $. $}
positive-in-var $a #Positive xX yY $. positive-in-symbol $a #Positive xX sg0 $.
positive-in-bot $a #Positive xX \bot $.
${ positive-in-imp.0 $e #Negative xX ph0 $.
   positive-in-imp.1 $e #Positive xX ph1 $.
   positive-in-imp   $a #Positive xX ( \imp ph0 ph1 ) $. $}
${ positive-in-app.0 $e #Positive xX ph0 $.
   positive-in-app.1 $e #Positive xX ph1 $.
   positive-in-app   $a #Positive xX ( \app ph0 ph1 ) $. $}
${ positive-in-exists.0 $e #Positive xX ph0 $.
   positive-in-exists   $a #Positive xX ( \exists x ph0 ) $. $}
${ positive-in-mu.0 $e #Positive xX ph0 $.
   positive-in-mu   $a #Positive xX ( \mu X ph0 ) $. $}
```

```
${ $d xX ph0 $. positive-disjoint $a #Positive xX ph0 $. $}
${ $d xX yY $. negative-in-var $a #Negative xX yY $. $}
negative-in-symbol $a #Negative xX sg0 $.
negative-in-bot $a #Negative xX \bot $.
${ negative-in-imp.0 $e #Positive xX ph0 $.
   negative-in-imp.1 $e #Negative xX ph1 $.
   negative-in-imp   $a #Negative xX ( \imp ph0 ph1 ) $. $}
${ negative-in-app.0 $e #Negative xX ph0 $.
   negative-in-app.1 $e #Negative xX ph1 $.
   negative-in-app   $a #Negative xX ( \app ph0 ph1 ) $. $}
${ negative-in-exists.0 $e #Negative xX ph0 $.
   negative-in-exists   $a #Negative xX ( \exists x ph0 ) $. $}
${ negative-in-mu.0 $e #Negative xX ph0 $.
   negative-in-mu   $a #Negative xX ( \mu X ph0 ) $. $}
${ $d xX ph0 $. negative-disjoint $a #Negative xX ph0 $. $}
${ $d xX yY $. fresh-in-var $a #Fresh xX yY $. $}
fresh-in-symbol $a #Fresh xX sg0 $. fresh-in-bot    $a #Fresh xX \bot $.
${ fresh-in-imp.0 $e #Fresh xX ph0 $. fresh-in-imp.1 $e #Fresh xX ph1 $.
   fresh-in-imp   $a #Fresh xX ( \imp ph0 ph1 ) $. $}
${ fresh-in-app.0 $e #Fresh xX ph0 $. fresh-in-app.1 $e #Fresh xX ph1 $.
   fresh-in-app   $a #Fresh xX ( \app ph0 ph1 ) $. $}
${ $d xX x $. fresh-in-exists.0 $e #Fresh xX ph0 $.
   fresh-in-exists $a #Fresh xX ( \exists x ph0 ) $. $}
fresh-in-exists-shadowed $a #Fresh x ( \exists x ph0 ) $.
${ $d xX X $. fresh-in-mu.0 $e #Fresh xX ph0 $.
   fresh-in-mu $a #Fresh xX ( \mu X ph0 ) $. $}
fresh-in-mu-shadowed $a #Fresh X ( \mu X ph0 ) $.
${ $d xX ph0 $. fresh-disjoint $a #Fresh xX ph0 $. $}
${ fresh-in-substitution.0 $e #Fresh xX ph1 $.
   fresh-in-substitution.1 $e #Substitution ph2 ph0 ph1 xX $.
   fresh-in-substitution $a #Fresh xX ph2 $. $}
${ fresh-after-substitution.0 $e #Fresh xX ph0 $.
   fresh-after-substitution.1 $e #Fresh xX ph1 $.
   fresh-after-substitution.2 $e #Substitution ph2 ph0 ph1 yY $.
   fresh-after-substitution $a #Fresh xX ph2 $. $}
substitution-var-same $a #Substitution ph0 xX ph0 xX $.
${ $d xX yY $. substitution-var-diff $a #Substitution yY yY ph0 xX $. $}
substitution-symbol $a #Substitution sg0 sg0 ph0 xX $.
substitution-bot    $a #Substitution \bot \bot ph0 xX $.
${ substitution-imp.0 $e #Substitution ph1 ph3 ph0 xX $.
   substitution-imp.1 $e #Substitution ph2 ph4 ph0 xX $.
   substitution-imp
   $a #Substitution ( \imp ph1 ph2 ) ( \imp ph3 ph4 ) ph0 xX $. $}
${ substitution-app.0  $e #Substitution ph1 ph3 ph0 xX $.
   substitution-app.1  $e #Substitution ph2 ph4 ph0 xX $.
   substitution-app
   $a #Substitution ( \app ph1 ph2 ) ( \app ph3 ph4 ) ph0 xX $. $}
substitution-exists-shadowed
   $a #Substitution ( \exists x ph1 ) ( \exists x ph1 ) ph0 x $.
${ $d xX x  $. $d y ph0 $.
   substitution-exists.0 $e #Substitution ph2 ph1 y x $.
   substitution-exists.1 $e #Substitution ph3 ph2 ph0 xX $.
   substitution-exists
   $a #Substitution ( \exists y ph3 ) ( \exists x ph1 ) ph0 xX $. $}
substitution-mu-shadowed $a #Substitution ( \mu X ph1 ) ( \mu X ph1 ) ph0 X $.
${ $d xX X  $. $d Y ph0 $.
   substitution-mu.0 $e #Substitution ph2 ph1 Y X $.
   substitution-mu.1 $e #Substitution ph3 ph2 ph0 xX $.
   substitution-mu   $a #Substitution ( \mu Y ph3 ) ( \mu X ph1 ) ph0 xX $. $}
substitution-identity $a #Substitution ph0 ph0 xX xX $.
${ yY-free-in-ph0 $e #Fresh yY ph0 $.
   ph1-definition $e #Substitution ph1 ph0 yY xX $.
   ${  substitution-fold.0   $e #Substitution ph2 ph1 ph3 yY $.
       substitution-fold     $a #Substitution ph2 ph0 ph3 xX $. $}
   ${  substitution-unfold.0 $e #Substitution ph2 ph0 ph3 xX $.
       substitution-unfold   $a #Substitution ph2 ph1 ph3 yY $. $} $}
${ substitution-inverse.0 $e #Fresh xX ph0 $.
   substitution-inverse.1 $e #Substitution ph1 ph0 xX yY $.
   substitution-inverse   $a #Substitution ph0 ph1 yY xX $. $}
${ substitution-fresh.0 $e #Fresh xX ph0 $.
   substitution-fresh $a #Substitution ph0 ph0 ph1 xX $. $}
application-context-var $a #ApplicationContext xX xX $.
${ $d xX ph1 $. application-context-app-left.0 $e #ApplicationContext xX ph0 $.
    application-context-app-left
    $a #ApplicationContext xX ( \app ph0 ph1 ) $. $}
${ $d xX ph0 $. application-context-app-right.0 $e #ApplicationContext xX ph1 $.
    application-context-app-right
    $a #ApplicationContext xX ( \app ph0 ph1 ) $. $}
notation-reflexivity $a #Notation ph0 ph0 $.
${ notation-symmetry.0 $e #Notation ph0 ph1 $.
   notation-symmetry   $a #Notation ph1 ph0 $. $}
${ notation-transitivity.0 $e #Notation ph0 ph1 $.
   notation-transitivity.1 $e #Notation ph1 ph2 $.
   notation-transitivity   $a #Notation ph0 ph2 $. $}
${ notation-positive.0 $e #Positive xX ph0 $.
   notation-positive.1 $e #Notation ph1 ph0 $.
   notation-positive   $a #Positive xX ph1 $. $}
${ notation-negative.0 $e #Negative xX ph0 $.
   notation-negative.1 $e #Notation ph1 ph0 $.
   notation-negative   $a #Negative xX ph1 $. $}
${ notation-fresh.0 $e #Fresh xX ph0 $.
   notation-fresh.1 $e #Notation ph1 ph0 $.
   notation-fresh   $a #Fresh xX ph1 $. $}
${ notation-substitution.0 $e #Substitution ph0 ph1 ph2 xX $.
   notation-substitution.1 $e #Notation ph3 ph0 $.
   notation-substitution.2 $e #Notation ph4 ph1 $.
   notation-substitution.3 $e #Notation ph5 ph2 $.
   notation-substitution   $a #Substitution ph3 ph4 ph5 xX $. $}
${ notation-notation.0 $e #Notation ph0 ph1 $.
   notation-notation.1 $e #Notation ph2 ph0 $.
   notation-notation.2 $e #Notation ph3 ph1 $.
   notation-notation   $p #Notation ph2 ph3 $=
   ( notation-transitivity notation-symmetry ) CADFABDEDBGIHH $. $}
${ notation-application-context.0 $e #ApplicationContext xX ph0 $.
   notation-application-context.1 $e #Notation ph1 ph0 $.
   notation-application-context   $a #ApplicationContext xX ph1 $. $}
${ notation-proof.0 $e |- ph0 $. notation-proof.1 $e #Notation ph1 ph0 $.
   notation-proof   $a |- ph1 $. $}
${ notation-imp.0 $e #Notation ph0 ph2 $. notation-imp.1 $e #Notation ph1 ph3 $.
   notation-imp $a #Notation ( \imp ph0 ph1 ) ( \imp ph2 ph3 ) $. $}
${ notation-app.0 $e #Notation ph0 ph2 $. notation-app.1 $e #Notation ph1 ph3 $.
   notation-app $a #Notation ( \app ph0 ph1 ) ( \app ph2 ph3 ) $. $}
${ notation-exists.0 $e #Notation ph0 ph1 $.
   notation-exists $a #Notation ( \exists x ph0 ) ( \exists x ph1 ) $. $}
${ notation-mu.0 $e #Notation ph0 ph1 $.
   notation-mu $a #Notation ( \mu X ph0 ) ( \mu X ph1 ) $. $}
$c \not $. not-is-pattern $a #Pattern ( \not ph0 ) $.
not-is-sugar $a #Notation ( \not ph0 ) ( \imp ph0 \bot ) $.
$c \or $. or-is-pattern $a #Pattern ( \or ph0 ph1 ) $.
or-is-sugar $a #Notation ( \or ph0 ph1 ) ( \imp ( \not ph0 ) ph1 ) $.
$c \and $. and-is-pattern $a #Pattern ( \and ph0 ph1 ) $.
and-is-sugar
  $a #Notation ( \and ph0 ph1 ) ( \not ( \or ( \not ph0 ) ( \not ph1 ) ) ) $.
proof-rule-prop-1 $a |- ( \imp ph0 ( \imp ph1 ph0 ) ) $.
proof-rule-prop-2 $a |- ( \imp ( \imp ph0 ( \imp ph1 ph2 ) )
                               ( \imp ( \imp ph0 ph1 ) ( \imp ph0 ph2 ) ) ) $.
proof-rule-prop-3 $a |- ( \imp ( \imp ( \imp ph0 \bot ) \bot ) ph0 ) $.
${ proof-rule-mp.0 $e |- ( \imp ph0 ph1 ) $.
   proof-rule-mp.1 $e |- ph0 $.
   proof-rule-mp   $a |- ph1 $. $}
${ proof-rule-exists.0 $e #Substitution ph0 ph1 y x $.
   proof-rule-exists   $a |- ( \imp ph0 ( \exists x ph1 ) ) $. $}
${ proof-rule-gen.0 $e |- ( \imp ph0 ph1 ) $.
   proof-rule-gen.1 $e #Fresh x ph1 $.
   proof-rule-gen   $a |- ( \imp ( \exists x ph0 ) ph1 ) $. $}
${ proof-rule-propagation-bot.0 $e #ApplicationContext xX ph0 $.
   proof-rule-propagation-bot.1 $e #Substitution ph1 ph0 \bot xX $.
   proof-rule-propagation-bot $a |- ( \imp ph1 \bot ) $. $}
${ proof-rule-propagation-or.0 $e #ApplicationContext xX ph0 $.
   proof-rule-propagation-or.1 $e #Substitution ph1 ph0 ( \or ph4 ph5 ) xX $.
   proof-rule-propagation-or.2 $e #Substitution ph2 ph0 ph4 xX $.
   proof-rule-propagation-or.3 $e #Substitution ph3 ph0 ph5 xX $.
   proof-rule-propagation-or $a |- ( \imp ph1 ( \or ph2 ph3 ) ) $. $}
${ proof-rule-propagation-exists.0 $e #ApplicationContext xX ph0 $.
   proof-rule-propagation-exists.1
     $e #Substitution ph1 ph0 ( \exists y ph3 ) xX $.
   proof-rule-propagation-exists.2 $e #Substitution ph2 ph0 ph3 xX $.
   proof-rule-propagation-exists.3 $e #Fresh y ph0 $.
   proof-rule-propagation-exists $a |- ( \imp ph1 ( \exists y ph2 ) ) $. $}
${ proof-rule-frame.0 $e #ApplicationContext xX ph0 $.
   proof-rule-frame.1 $e #Substitution ph1 ph0 ph3 xX $.
   proof-rule-frame.2 $e #Substitution ph2 ph0 ph4 xX $.
   proof-rule-frame.3 $e |- ( \imp ph3 ph4 ) $.
   proof-rule-frame   $a |- ( \imp ph1 ph2 ) $. $}
${ proof-rule-prefixpoint.0 $e #Substitution ph0 ph1 ( \mu X ph1 ) X $.
   proof-rule-prefixpoint   $a |- ( \imp ph0 ( \mu X ph1 ) ) $. $}
${ proof-rule-kt.0 $e #Substitution ph0 ph1 ph2 X $.
   proof-rule-kt.1 $e |- ( \imp ph0 ph2 ) $.
   proof-rule-kt   $a |- ( \imp ( \mu X ph1 ) ph2 ) $. $}
${ proof-rule-set-var-substitution.0 $e #Substitution ph0 ph1 ph2 X $.
   proof-rule-set-var-substitution.1 $e |- ph1 $.
   proof-rule-set-var-substitution   $a |- ph0 $. $}
proof-rule-existence $a |- ( \exists x x ) $.
${ proof-rule-singleton.0 $e #ApplicationContext xX ph0 $.
   proof-rule-singleton.1 $e #ApplicationContext yY ph1 $.
   proof-rule-singleton.2 $e #Substitution ph3 ph0 ( \and x ph2 ) xX $.
   proof-rule-singleton.3 $e #Substitution ph4 ph1 ( \and x ( \not ph2 ) ) yY $.
   proof-rule-singleton $a |- ( \not ( \and ph3 ph4 ) ) $. $}
```

Chapter 8: PROOF-CERTIFYING PROGRAM EXECUTION

Our vision is that of an ideal language framework, as shown in Figure 1.1, where programming language designers only need to write formal definitions of their languages, and all language tools are automatically generated by the framework. The $\mathbb{K}$ framework, as discussed in Section 2.15, pursues the above vision by providing a simple and intuitive front-end language (i.e., a meta-language) for defining programming languages. $\mathbb{K}$ also provides a set of language-agnostic (also called language-independent or language-parametric) tools, including a parser, an interpreter, a deductive verifier, and a program equivalence checker [3, 4]. These tools can be instantiated by the formal semantics of any given programming language.

What is missing in $\mathbb{K}$ is the ability to generate machine-checkable correctness certificates. Currently, $\mathbb{K}$ has over 500,000 lines of code written in 4 programming languages, with new code committed to the code base on a weekly basis. The $\mathbb{K}$ code base includes complex data structures, algorithms, optimizations, and heuristics to support the various features that are needed for defining the formal semantics of programming languages. For example, $\mathbb{K}$ uses BNF grammars for defining formal language syntax, constructors and terms for defining computation configurations, rewrite rules for defining operational semantics, and strictness and contexts for defining evaluation orders. All the above make it challenging to formally verify the correctness of $\mathbb{K}$.

The objective of this chapter and Chapter 9 is to propose a practical approach to establishing the correctness of $\mathbb{K}$ via proof generation. The idea is as follows. For any programming language L defined in $\mathbb{K}$, we translate its formal semantics into an AML theory Γ^L. For any computation and formal analysis task (e.g., executing a program or verifying the functional correctness of a program) carried out using $\mathbb{K}$, we encode its correctness as an AML pattern φ_{task}. Then, the correctness of $\mathbb{K}$ is reduced to proving the following AML theorem:

$$\Gamma^L \vdash \varphi_{task}$$

Proof generation is the process of generating a formal proof for the above theorem, for the given L and φ_{task}. The outcome of proof generation is a *proof object* for $\Gamma^L \vdash \varphi_{task}$ that can be directly checked by the AML proof checker in Section 7.4. This way, the correctness of $\mathbb{K}$ is reduced to the correctness of proof checking.

In this chapter we focus on proof generation for program execution. For any execution trace, we generate a corresponding AML proof object that justifies its correctness. Since the correctness certificate is generated for each execution trace, we achieve proof-certifying program execution. In Chapter 9, we apply the same idea to achieve proof-certifying formal verification.

8.1 OVERVIEW

Our approach to proof-certifying program execution consists of four components: (1) AML as a logical foundation of $\mathbb{K}$; (2) proof hints; (3) the proof generation procedures; and (4) the AML proof checker in Section 7.4. We give an overview of these components below.

AML serves as the logical foundation of our proof generation process and also of $\mathbb{K}$. By that, we mean that any programming language L defined in K is translated to an AML theory Γ^L, which, roughly speaking, consists of symbols that represent the formal syntax of L and axioms that specify its formal semantics. Program execution is specified by the following theorem:

$$\Gamma^L \vdash \varphi_{\mathit{init}} \Rightarrow_{\mathit{exec}} \varphi_{\mathit{final}} \tag{8.1}$$

where φ_{init} and φ_{final} are patterns representing the initial and final states, respectively. The operation $\Rightarrow_{\mathit{exec}}$ is defined as a notation, i.e., $\varphi_{\mathit{init}} \Rightarrow_{\mathit{exec}} \varphi_{\mathit{final}} \equiv \varphi_{\mathit{init}} \to \diamond\varphi_{\mathit{final}}$, where $\diamond\varphi_{\mathit{final}} \equiv \mu X \,.\, \varphi_{\mathit{final}} \vee \bullet X$ is the "eventually" operator in Section 5.7.

A *proof hint* consists of the necessary information that $\mathbb{K}$ should give to the proof generation procedures to help generate proof objects. For program execution, a proof hint includes the following information:

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

Given a proof hint, the proof generation procedure for program execution calls a subprocedure to generate the proof objects for all the one-step execution steps, i.e., $\Gamma^L \vdash \varphi_i \Rightarrow_{\mathit{exec}} \varphi_{i+1}$ for all i. For each of the sub-goal, we generate its proof object by further decomposing it into applying a rewrite rule and applying simplification rules, which can be further decomposed into applying substitution, equational reasoning, etc. Once all the sub-goals are proved, we put together all the generated sub-proof objects and output the final proof object. The generated proof objects can be automatically checked by the AML proof checker in Section 7.4.

To sum up, our approach to proof-certifying program execution is based on $\mathbb{K}$ and its logical foundation AML. Programming language semantics defined in $\mathbb{K}$ are translated to

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

Figure 8.1: Running Example `TWO-COUNTERS`.

AML theories. Program execution can be formalized as AML theorems, whose proofs are automatically generated and checked. The key characteristics of our approach are that:

1. it is faithful to the actual implementation of $\mathbb{K}$ because proof certificates are generated from proof hints, which include all the intermediate snapshots and the actual rewriting information, provided by $\mathbb{K}$;

2. it is practical because correctness certificates are generated for each execution case on a case-by-case basis, avoiding the verification of the entire $\mathbb{K}$;

3. it is trustworthy because the correctness certificates can automatically checked by a proof checker.

8.2 A RUNNING EXAMPLE

We use a simple example as shown in Figure 8.1 to explain our proof generation procedures. The semantics `TWO-COUNTERS` defines a state machine with two counters. A computation configurations is a pair $\langle m, n\rangle$ and its semantics is given by the following rewrite rule:

$$\langle m, n\rangle \Rightarrow \langle m-1, n+m\rangle \qquad \text{if } m > 0 \tag{8.2}$$

In each execution step, `TWO-COUNTERS` adds n by m and reduces m by 1. Starting from the initial state $\langle m, 0\rangle$, `TWO-COUNTERS` carries out m execution steps and terminates at the final state $\langle 0, m(m+1)/2\rangle$, where $m(m+1)/2 = m + (m-1) + \cdots + 1$. The following shows a concrete program execution trace of `TWO-COUNTERS` starting from the initial state $\langle 100, 0\rangle$:

$$\langle 100, 0\rangle, \langle 99, 100\rangle, \langle 98, 199\rangle, \ldots, \langle 1, 5049\rangle, \langle 0, 5050\rangle \tag{8.3}$$

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

1. Prepare the initial state $\langle 100, 0\rangle$ in a source file, say `100.two-counters`.

2. Compile TWO-COUNTERS into an AML theory (discussed in Section 8.3);

3. Use the 𝕂 execution tool krun and pass the source file to it:

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

The option --depth N tells 𝕂 to execute for N steps and output the corresponding intermediaate snapshot. By letting N be 1, 2, . . . , we collect all the intermediate snapshots in Equation (8.3).

The proof hint of Equation (8.3) includes the rewriting information for each execution step, i.e., the rewrite rule that is applied and the corresponding substitution. In TWO-COUNTERS, there is only one rewrite rule, and the substitution can be easily obtained by pattern matching, where we simply match the snapshot with the left-hand side of the rewrite rule.

Note that we regard 𝕂 as a "black box". We are not interested in its complex internal algorithms. Instead, we hide such complexity by letting 𝕂 generate proof hints. This way, we create a separation of concerns between 𝕂 and proof generation. 𝕂 can aim at optimizing the performance of the auto-generated language tools, without making proof generation more complex.

8.3 TRANSLATING 𝕂 TO AML

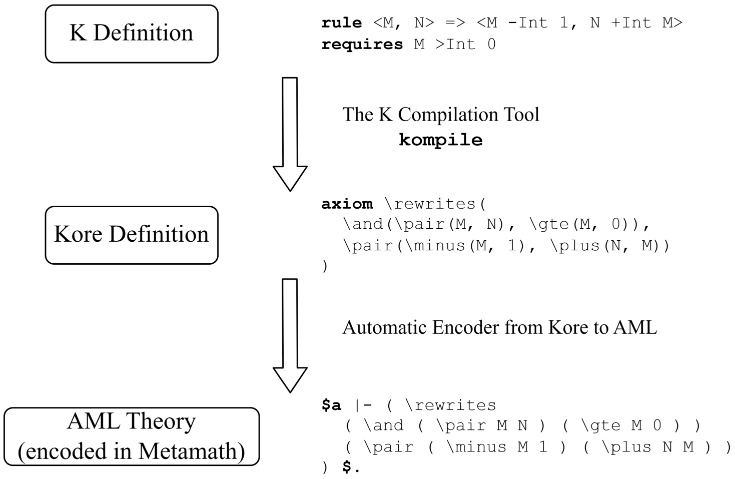
To compile programming languages semantics in 𝕂 to AML theories, we use the existing 𝕂 compilation tool kompile. The tool kompile translates a 𝕂 semantics into an AML theory written in a formal language called Kore, which is based on AML extended with the theories of equality, sorts, and rewriting. To formalize the compiled Kore definitions in proof objects, we first formalize the theories of equality, sorts, and rewriting and then translate Kore definitions into AML axioms, as shown in Figure 8.2.

Phase-1 translation is from 𝕂 to Kore, where we pass two-counters.k to kompile:

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

The result is a compiled Kore definition two-counters.kore. Figure 8.2 shows an example auto-generated Kore axiom that corresponds to the rewrite rule in Equation (8.2). As we can see, Kore is at a much lower-level than 𝕂, where the programming language concrete syntax and 𝕂's front-end syntax are parsed and replaced by the abstract syntax trees, represented by the constructor terms.

Phase-2 translation is from Kore to AML. We develop an automatic encoder that translates Kore syntax into AML patterns. Since Kore is essentially the theory of equality, sorts, and



```
rule <M, N> => <M -Int 1, N +Int M>
requires M >Int 0
```

```
axiom \rewrites(
  \and(\pair(M, N), \gte(M, 0)),
  \pair(\minus(M, 1), \plus(N, M))
)
```

```
$a |- ( \rewrites
  ( \and ( \pair M N ) ( \gte M 0 ) )
  ( \pair ( \minus M 1 ) ( \plus N M ) )
) $.
```

Figure 8.2: Two-Phase Translation from $\mathbb{K}$ to AML via Kore

rewriting, we define the syntactic constructs of the Kore language as AML notation and theories, using the mechanisms introduced in Section 7.4.

8.4 GENERATING PROOFS FOR ONE-STEP EXECUTIONS

The key step in our approach is to generate proof objects for *one-step executions.* which are then put together to build the final proof objects for an entire execution trace using the transitivity of the rewriting relation. Thus, we focus on the proof generation procedures for one-step executions.

8.4.1 Problem formulation

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

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

where t_k and s_k are the left- and right-hand sides of the rewrite rule, respectively, and p_k is the rewriting condition. Consider an execution trace $\varphi_0, \varphi_1, \ldots, \varphi_n$ where $\varphi_0, \ldots, \varphi_n$ are intermediate snapshots. We let $\mathbb{K}$ generate the following proof hint:

$$\Theta \equiv (k_0, \theta_0), \ldots, (k_{n-1}, \theta_{n-1}) \tag{8.4}$$

where for each $0 \leq i < n$, k_i denotes the rewrite rule that is applied on φ_i $(1 \leq k_i \leq K)$ and θ_i denotes the corresponding substitution such that $t_{k_i}\theta_i = \varphi_i$. For example, the rewrite rule of `TWO-COUNTERS`, restated below:

$$\langle m, n\rangle \Rightarrow_{exec} \langle m-1, n+m\rangle \quad \text{if } m > 0$$

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

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

As we can see, there are two types of proof goals: one for applying rewrite rules and one for applying simplification rules. We discuss their proof generation procedures in the following.

8.4.2 Applying rewrite rules

The main steps in proving $\Gamma^L \vdash \varphi_i \Rightarrow s_{k_i}\theta_i$ are to instantiate the rewrite rule $t_{k_i} \wedge p_{k_i} \Rightarrow s_{k_i}$ by the substitution

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

in the proof hint, and then show that the (instantiated) rewriting condition $p_{k_i}\theta_i$ holds. Here, $x_1, \ldots, x_m$ are the variables that occur in the rewrite rule and $c_1, \ldots, c_m$ are terms by which we instantiate the variables. For (1), we need to first prove the following lemma, called (Functional Substitution) in Figure 2.11, which states that ∀-quantification can be instantiated by functional patterns:

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

Intuitively, the premise $\exists y_1 \,.\, \varphi_1 = y_1$ states that φ_1 is a functional pattern because it equals to some element y_1. If Θ in Equation (8.4) is the correct proof hint, θ_i is the correct substitution and thus $t_{k_i}\theta_i \equiv \varphi_i$. Therefore, to prove the original proof goal for one-step execution, i.e. $\Gamma^L \vdash \varphi_i \Rightarrow s_{k_i}\theta_i$, we only need to prove that $\Gamma^L \vdash p_{k_i}\theta_i$, i.e., the rewriting condition p_{k_i} holds under θ_i. This is done by simplifying $p_{k_i}\theta_i$ to $\top$, discussed below.

8.4.3 Applying simplification rules

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

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

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

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

8.5 EVALUATION

In this section, we evaluate the performance of our implementation and discuss the experiment results, summarized in Table 8.1. We use two sets of benchmarks. The first is our running example `TWO-COUNTERS` with different inputs (10, 20, 50, and 100). The second is REC [112], which is a popular performance benchmark for rewriting engines. We evaluate both the performance of proof generation and that of proof checking. Our implementation can be found in [113]. The main takeaways of our experiments are the following:

Table 8.1: Proof Generation and Proof Checking Performance: Program Execution

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

1. Proof checking is efficient and takes a few seconds; in particular, the task-specific checking time is often less than one second (see the "task" column in Table 8.1).

2. Proof generation is slower and takes several minutes.

3. Proof objects are huge, often of millions LOC (wrapped at 80 characters).

We measure the proof generation time as the time to generate complete proof objects following the proof generation procedures in Section 8.4, from the compiled Kore definitions and proof hints. As shown in Table 8.1, proof generation takes around 17–406 seconds on the benchmarks, and the average is 107 seconds. Proof generation can be divided into two parts: that of the language semantics Γ^L and that of the (one-step and multi-step) program executions. Both parts are shown in Table 8.1 under columns "sem" and "rewrite", respectively. For the same language, the time to generate language semantics Γ^L is the same (up to experimental error). The time for executions is linear to the number of steps.

Proof checking is efficient and takes a few seconds on our benchmarks. We can divide the proof checking time into two parts: that of the logical foundation and that of the actual program execution tasks. Both parts are shown in Table 8.1 under columns "logic" and "task". The "logic" part includes formalization of AML and its basic theories, and thus is fixed for any programming language and program and has the same proof checking time (up to experimental error). The "task" part includes the language semantics and proof objects for the one-step and multi-step executions. Therefore, the time to check the "task" part is a more valuable and realistic measure, and according to our experiments, it is often less than 1 second, making it acceptable in practice.

Note that the time for "task-specific" proof checking is roughly the same as the time for 𝕂 to parse and execute the program. There is no significant performance difference on our benchmarks between running the programs directly in 𝕂 and checking the proof objects. Furthermore, there exists much potential to optimize the performance of proof checking and make it even faster than program execution. For example, proof checking is an embarrassingly parallel problem, because each meta-theorems can be proof-checked entirely independently. We can thus further reduce the proof checking time by running multiple instances of the proof checker in parallel.

Chapter 9: PROOF-CERTIFYING FORMAL VERIFICATION

We push the idea in Chapter 8 further and apply it to achieve proof-certifying formal verification. We first review the verification algorithm (Algorithm 9.1) that automates the reachability proof rules in Figure 2.12 in Section 9.1. Then, we describe the proof generation procedures for symbolic execution (Section 9.2), pattern subsumption (Section 9.3), and coinductive reasoning (Section 9.4). We discuss the interesting implementation details in Section 9.6 and show evaluation results in Section 9.5.

9.1 OVERVIEW

We show the language-agnostic verification algorithm of $\mathbb{K}$ in Algorithm 9.1, which is an optimized implementation of the reachability proof rules in Figure 2.12. The input R is a set of reachability claims to be verified, including the necessary invariant claims. The algorithm consists of two procedures: `proveAllClaims` and `proveOneClaim`. The first calls the latter on every input claim. The procedure `proveOneClaim` starts by checking the subsumption $\Gamma^L \vdash \varphi \to \varphi'$. If it holds, then the claim $\varphi \Rightarrow_{\text{reach}} \varphi'$ is trivially true. If the direct subsumption is false, we perform symbolic execution for one step from φ to get a set Q of all its successors. Both `successors` (Line 8) and $\texttt{successors}_R$ (Line 12) calculate all the successors of a given configuration. `successors` uses only the formal semantics in Γ^L while $\texttt{successors}_R$ uses both the semantic rules and the claims in R. This is sound because at least one real semantic step has been made in Line 8. If $Q \neq \emptyset$, the algorithm nondeterministically chooses a frontier pattern ψ_{front} from Q and checks whether ψ_{front} satisfies φ'. If yes, the verification succeeds (Line 11). Otherwise, the algorithm symbolically executes ψ_{front} and continues with its successors (Line 12), following both the semantic rules and the claims in R. This is sound because in Line 8, before the `while` loop, we have computed the successors of φ using only the semantic rules. Immediately after that, when we entered the loop for the first time, we chose one successor of φ, say φ_s (Line 10). Therefore, we have $\Gamma^L \vdash \varphi \Rightarrow^+_{\text{reach}} \varphi_s$. Since at least one execution step has been made, the (Transitivity) rule in Figure 2.12 moves all the circularity claims (i.e., the claims in R) to the axiom set so they can be used as semantic axioms in computing further successors (Line 12).

In this work we only consider verifying reachability claims on one path, called one-path reachability. The procedure `proveOneClaim` nondeterministically chooses a frontier pattern ψ_{front} from all the possible successors in Q (see Line 10), which amounts to looking for the one execution path that satisfies the reachability claim. Therefore, `proveOneClaim` is

Algorithm 9.1: Algorithm for Proving One-Path Reachability Claims

```
1  procedure proveAllClaims($R$)
2      foreach $\varphi \Rightarrow_{reach} \varphi' \in R$ do
3          if proveOneClaim($R$, $\varphi \Rightarrow_{reach} \varphi'$) = failure then return failure;
4      return success;
5  // a nondeterministic algorithm for proving one reachability claim
6  procedure proveOneClaim($R$, $\varphi \Rightarrow_{reach} \varphi'$)
7      if $\Gamma^L \vdash \varphi \to \varphi'$ then return success;
8      $Q$ :− successors($\varphi$);
9      while $Q \neq \emptyset$ do
10         $\psi_{\text{front}}$ :− choose($Q$);    // a nondeterministic choice
11         if $\Gamma^L \vdash \psi_{\text{front}} \to \varphi'$ then return success;
12         else $Q$ :− successors$_R$($\psi_{\text{front}}$);
13     return failure;
```

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

Our goal is to generate proof objects for Algorithm 9.1. For clarity, we divide it into three proof generation procedures:

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

We discuss these proof generation procedures in the following.

9.2 GENERATING PROOFS FOR SYMBOLIC EXECUTION

We use Γ^L to denote the AML theory of the formal semantics of a language L. Consider the following $\mathbb{K}$ language definition, which consists of K (conditional) rewrite rules:

$$\{lhs_k \wedge q_k \Rightarrow^1_{exec} rhs_k \mid k = 1, 2, \ldots, K\} \subseteq \Gamma^L$$

where lhs_k represents the left-hand side of the rewrite rule, rhs_k represents the right-hand side, and q_k denotes the rewriting condition. Unconditional rules can be regarded as conditional rules where q_k is $\top$. The notation $\Rightarrow^1_{exec}$ stands for one-step execution, defined as $\varphi_1 \Rightarrow^1_{exec} \varphi_2 \equiv \varphi_1 \to \bullet\varphi_2$.

In symbolic execution, program configurations often appear with their corresponding path conditions. We represent them as $t \wedge p$, where t is a configuration and p is a logical constraint/predicate over the free variables of t. We call such patterns constrained terms. Constrained terms are AML patterns.

Unlike concrete execution, symbolic execution can create branches. Therefore, we formulate proof generation for symbolic execution as follows. The input is an initial constrained term $t \wedge p$ and a list of final constrained terms $t_1 \wedge p_1, \ldots, t_n \wedge p_n$, which are returned by $\mathbb{K}$ as the result(s) of symbolic executing t under the condition p. Each $t_i \wedge p_i$ represents one possible execution trace. Our goal is to generate a proof for the following goal:

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

In other words, here we are certifying the correctness of the `successors` (and $\texttt{successors}_R$) methods used by Algorithm 9.1, by proving that $\Gamma^L \vdash \varphi \Rightarrow_{exec} \texttt{successors}(\varphi)$, which further implies $\Gamma^L \vdash \varphi \Rightarrow_{\text{reach}} \texttt{successors}(\varphi)$.

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

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

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

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

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

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

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

And for the remainder branch, we need to prove

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

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

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

We explain these phases in the following. Note that we need many lemmas about the program execution relation "$\Rightarrow_{\mathit{exec}}$" when we generate proof objects for symbolic execution. The most important and relevant lemmas are stated explicitly in this paper. In total, 196 new lemmas are formally encoded, and their proofs have been completely worked out based on the Metamath formalization of the proof system [113, 114], as a part of the new contribution of the paper. These lemmas can be easily reused for future development.

9.2.1 Phase 1: Proving (Branch$_{j,l}$) and (Remainder$_j$)

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

$$\mathit{lhs}_{k_{j,l}} \wedge q_{k_{j,l}} \Rightarrow^1_{\mathit{exec}} \mathit{rhs}_{k_{j,l}}$$

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

$$\Gamma^L \vdash \mathit{lhs}_{k_{j,l}}\theta_{j,l} \wedge q_{k_{j,l}}\theta_{j,l} \Rightarrow^1_{\mathit{exec}} \mathit{rhs}_{k_{j,l}}\theta_{j,l} \tag{9.1}$$

where we use $t\theta$ to denote the result of applying the substitution θ to t. Note that $q_{k_{j,l}}\theta_{j,l}$ is a predicate on the free variables of Equation (9.1) that holds on the left-hand side, by propositional reasoning, it also holds on the right-hand side. Therefore, we prove that:

$$\Gamma^L \vdash \mathit{lhs}_{k_{j,l}}\theta_{j,l} \wedge q_{k_{j,l}}\theta_{j,l} \Rightarrow^1_{\mathit{exec}} \mathit{rhs}_{k_{j,l}}\theta_{j,l} \wedge q_{k_{j,l}}\theta_{j,l} \tag{9.2}$$

To proceed, we need the following lemma:

Lemma 9.1 ($\Rightarrow^1_{\mathit{exec}}$ Consequence)**.**

$$\frac{\Gamma^L \vdash \varphi \to \varphi' \quad \Gamma^L \vdash \varphi' \Rightarrow^1_{\mathit{exec}} \psi' \quad \Gamma^L \vdash \psi' \to \psi}{\Gamma^L \vdash \varphi \Rightarrow^1_{\mathit{exec}} \psi}$$

Intuitively, Lemma 9.1 allows us to strengthen the left-hand side and/or weaken the right-hand side of an execution relation. Using Lemma 9.1, and by comparing our proof goal (Branch$_{j,l}$) with Equation (9.2), we only need to prove the following two implications between constrained terms, which we call *subsumptions*:

$$\underbrace{\Gamma^L \vdash \left(t_j^{\text{hint}} \wedge p_{j,l}^{\text{hint}}\right) \to \left(\mathit{lhs}_{k_{j,l}}\theta_{k_{j,l}} \wedge q_{k_{j,l}}\theta_{k_{j,l}}\right)}_{\text{left-hand side strengthening}} \quad \underbrace{\Gamma^L \vdash \left(\mathit{rhs}_{k_{j,l}}\theta_{k_{j,l}} \wedge q_{k_{j,l}}\theta_{k_{j,l}}\right) \to \left(t_{j,l}^{\text{hint}} \wedge p_{j,l}^{\text{hint}}\right)}_{\text{right-hand side weakening}}$$

These subsumption proofs are common in our proof generation procedure (e.g. (Remainder$_j$) is also a subsumption). We elaborate on subsumption proofs in Section 9.3.

9.2.2 Phase 2: Proving (Step$_j$)

We combine the proofs for each branch and the remainder as follows:

$$\Gamma^L \vdash t_j^{\text{hint}} \wedge p_{j,1}^{\text{hint}} \Rightarrow_{exec}^1 t_{j,1}^{\text{hint}} \wedge p_{j,1}^{\text{hint}} \tag{Branch$_{j,1}$}$$

$$\vdots$$

$$\Gamma^L \vdash t_j^{\text{hint}} \wedge p_{j,l_j}^{\text{hint}} \Rightarrow_{exec}^1 t_{j,l_j}^{\text{hint}} \wedge p_{j,l_j}^{\text{hint}} \tag{Branch$_{j,l_j}$}$$

$$\Gamma^L \vdash t_j^{\text{hint}} \wedge p_j^{\text{rem}} \to t_j^{\text{rem}} \wedge p_j^{\text{rem}} \tag{Remainder$_j$}$$

Note that our proof goal (Step$_j$) uses "$\Rightarrow_{exec}$", while the above use either one-step execution ("$\Rightarrow_{exec}^1$") or implication ("$\to$"). The following lemma allows us to turn one-step execution and implication (i.e. "zero-step execution") into the reflexive-transitive execution relation "$\Rightarrow_{exec}$":

Lemma 9.2 ($\Rightarrow_{exec}$ Introduction)**.**

$$\frac{\Gamma^L \vdash \varphi \to \psi}{\Gamma^L \vdash \varphi \Rightarrow_{exec} \psi} \qquad \frac{\Gamma^L \vdash \varphi \Rightarrow_{exec}^1 \psi}{\Gamma^L \vdash \varphi \Rightarrow_{exec} \psi}$$

Then, we need to verify that the disjunction of all path conditions in the branches (including the remainder) is implied from the initial path condition:

$$\Gamma^L \vdash p_j^{\text{hint}} \to p_{j,1}^{\text{hint}} \vee \cdots \vee p_{j,l_j}^{\text{hint}} \vee p_j^{\text{rem}} \tag{9.3}$$

The above implication includes only logical constraints and no configuration terms, and thus involves only domain reasoning. Therefore, we translate it into an equivalent FOL formula and delegate it to SMT solvers, such as Z3 [97].

From Equation (9.3), we can prove that the left-hand side of (Step$_j$), $t_j^{\text{hint}} \wedge p_j^{\text{hint}}$, can be broken down into $l_j + 1$ branches by propositional reasoning:

$$\Gamma^L \vdash \left(t_j^{\text{hint}} \wedge p_j^{\text{hint}}\right) \to \left(t_j^{\text{hint}} \wedge p_{j,1}^{\text{hint}}\right) \vee \ldots \vee \left(t_j^{\text{hint}} \wedge p_{j,l_j}^{\text{hint}}\right) \vee \left(t_j^{\text{hint}} \wedge p_j^{\text{rem}}\right) \tag{9.4}$$

Note that the right-hand side of Equation (9.4) is exactly the disjunction of all the left-hand sides of (Branch$_{j,l}$) and (Remainder$_j$). Therefore, to prove the proof goal (Step$_j$), we use the following lemma, which allows us to combine the executions in different branches into one (we will also need a consequence rule for $\Rightarrow_{exec}$ like Lemma 9.1, which is derivable from Lemmas 9.1 and 9.2):

Lemma 9.3 ($\Rightarrow_{exec}$ Merge)**.**

$$\frac{\Gamma^L \vdash \varphi_1 \Rightarrow_{exec} \psi_1 \quad \ldots \quad \Gamma^L \vdash \varphi_n \Rightarrow_{exec} \psi_n}{\Gamma^L \vdash \bigvee_{i=1}^{n} \varphi_i \Rightarrow_{exec} \bigvee_{i=1}^{n} \psi_i}$$

9.2.3 Phase 3: Proving (Goal)

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

$$\Gamma^L \vdash (t \wedge p) \Rightarrow_{exec} (t \wedge p) \tag{9.5}$$

Then, we repeatedly apply the following steps to symbolically execute the right-hand side of Equation (9.5), until it becomes the same as the right-hand side of (Goal):

1. Suppose we have obtained a proof object for

$$\Gamma^L \vdash (t \wedge p) \Rightarrow_{exec} \left(t_1^{\text{im}} \wedge p_1^{\text{im}}\right) \vee \cdots \vee \left(t_m^{\text{im}} \wedge p_m^{\text{im}}\right) \tag{9.6}$$

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

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

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

   such that $t_j^{\text{hint}} \wedge p_j^{\text{hint}} \equiv t_i^{\text{im}} \wedge p_i^{\text{im}}$, for some intermediate constrained term $t_i^{\text{im}} \wedge p_i^{\text{im}}$. Without loss of generality, let us assume that $i = 1$, i.e., the first intermediate constrained term $t_1^{\text{im}} \wedge p_1^{\text{im}}$ can be rewritten/executed using (Step$_j$).

3. Symbolically execute $t_1^{\text{im}} \wedge p_1^{\text{im}}$ in Equation (9.6) for one step by applying (Step$_j$), and obtain the following proof:

$$\Gamma^L \vdash (t \wedge p) \Rightarrow_{exec} \underbrace{\left(t_{j,1}^{\text{hint}} \wedge p_{j,1}^{\text{hint}}\right) \vee \cdots \vee \left(t_{j,l_j}^{\text{hint}} \wedge p_{j,l_j}^{\text{hint}}\right) \vee \left(t_j^{\text{rem}} \wedge p_j^{\text{rem}}\right)}_{\text{right-hand side of (Step}_j\text{)}}$$
$$\vee \underbrace{\left(t_2^{\text{im}} \wedge p_2^{\text{im}}\right) \vee \ldots \vee \left(t_m^{\text{im}} \wedge p_m^{\text{im}}\right)}_{\text{same as Equation (9.6)}}$$

Finally, after all symbolic execution steps are applied, we check if the resulting proof goal is the same as (Goal), potentially after permuting the disjuncts on the right-hand side. If yes, then the proof generation method succeeds and we generate a proof certificate for (Goal). Otherwise, the proof generation method fails, indicating potential mistakes made by $\mathbb{K}$'s symbolic execution engine.

9.3 GENERATING PROOFS FOR PATTERN SUBSUMPTION

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

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

To prove the first sub-goal $\Gamma^L \vdash p \to p'$, we note that both p and p' are logical constraints. Therefore, its proof is delegated to external SMT solvers. To prove the second sub-goal $\Gamma^L \vdash p \to (t = t')$, we first try an SMT solver with all constructors abstracted to uninterpreted functions. If the SMT solver proves the goal with such abstraction, our proof generation method succeeds. Otherwise, we break down t and t' into sub-terms. Specifically, if $t \equiv f(t_1, \ldots, t_n)$ and $t' \equiv f(t'_1, \ldots, t'_n)$, we reduce the sub-goal into a set of goals:

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

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

Our method here for pattern subsumption is incomplete but covers most simplifications done by $\mathbb{K}$. Generally speaking, it is undecidable to prove such subsumptions as it requires to prove first-order theorems in an initial algebra of an equational/algebraic specification. However, there exist techniques that are shown to be effective in automating inductive theorem proving, such as Maude ITP [58], which can be integrated by our work in the future.

9.4 GENERATING PROOFS FOR COINDUCTION

Recall that the verification algorithm (Algorithm 9.1) performs symbolic execution from the left-hand side of each claim until all branches are subsumed by the right-hand side.

While the proof generation procedures in previous sections Sections 9.2 and 9.3 can cover symbolic execution already, the missing part is line 12 in Algorithm 9.1, where we apply not the semantic rules but the claims in R to perform symbolic execution, which forms a circular argument. Our purpose is to generate proof objects that justify the soundness of such circular reasoning, by showing that the algorithm is performing a coinduction on the (potentially infinite) execution trace.

We start with the simplest case when R has only one claim $\varphi \Rightarrow_{\text{reach}} \psi$. We assume that we have already rewritten φ to some intermediate configuration φ' using at least one steps (so logically speaking, the set of claims $R = \{\varphi \Rightarrow_{\text{reach}} \psi\}$ has been flushed to the reachability logic axiom set by (Transitivity) in Figure 2.12):

$$\Gamma^L \vdash \varphi \Rightarrow^{+}_{\text{reach}} \varphi' \tag{9.7}$$

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

$$\Gamma^L \vdash \Box(\forall \mathit{freeVar}(\varphi, \psi) \,.\, \varphi \Rightarrow_{\text{reach}} \psi) \to \varphi' \Rightarrow_{\text{reach}} \varphi'' \tag{9.8}$$

where $\mathit{freeVar}(\varphi, \psi)$ is the set of all free variables in φ and ψ. Intuitively, we instantiate all the free variables using the substitution specified by the proof hint, where φ'' is the result of applying the claim $\varphi \Rightarrow_{\text{reach}} \psi$ as a regular semantic rule on φ'. Recall that Equation (9.8) is the encoding of the reachability judgment $\{\varphi \Rightarrow_{\text{reach}} \psi\} \vdash^{\mathit{reach}}_{\emptyset} \varphi' \Rightarrow \varphi''$.

Now, we apply (Transitivity) to Equations (9.7) and (9.8) and obtain the proof object for

$$\Gamma^L \vdash \circ\Box(\forall \mathit{freeVar}(\varphi, \psi) \,.\, \varphi \Rightarrow_{\text{reach}} \psi) \to \varphi \Rightarrow^{+}_{\text{reach}} \varphi''$$

which is the encoding of the reachability judgment $\vdash^{\mathit{reach}}_{\{\varphi \Rightarrow_{\text{reach}} \psi\}} \varphi \Rightarrow \varphi''$, where $\varphi \Rightarrow_{\text{reach}} \psi$ belongs to the circularity set. Then, we reuse the proof generation procedure in Section 9.2 to generate the proof object for the symbolic execution of φ'', except that now there is an additional premise $\circ\Box(\forall \mathit{freeVar}(\varphi, \psi) \,.\, \varphi \Rightarrow_{\text{reach}} \psi)$ that encodes the semantics of circularity.

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

$$\Gamma^L \vdash \circ\Box(\forall \mathit{freeVar}(\varphi, \psi) \,.\, \varphi \Rightarrow_{\text{reach}} \psi) \to \varphi \Rightarrow_{\text{reach}} \psi$$

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

Generally speaking, Algorithm 9.1 supports proving n claims at the same time, i.e., $R = \{\varphi_1 \Rightarrow_{\text{reach}} \psi_1, \ldots, \varphi_n \Rightarrow_{\text{reach}} \psi_n\}$, where the proof of each claim could arbitrarily invoke

Table 9.1: Proof Generation and Proof Checking Performance: Formal Verification

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

the other claims as coinduction hypotheses. This is called *set circularity*, which is derivable in reachability logic (see [115, Lemma 5])

$$(\textsc{Set Circularity}) \qquad \frac{A \vdash_R^{\mathit{reach}} \varphi \Rightarrow \psi \quad \text{for all } (\varphi \Rightarrow \psi) \in R}{A \vdash_\emptyset^{\mathit{reach}} \varphi \Rightarrow \psi \quad \text{for all } (\varphi \Rightarrow \psi) \in R}$$

Here, all the claims in R are simultaneously added to the circularity set, featuring a mutual coinduction among all the coinduction hypotheses. Our current implementation does not support (Set Circularity) in its full generality. We assume that the proof of each claim only invokes itself as the coinduction hypothesis. This is not a restriction in theory because using [115, Lemma 5], any proof using (Set Circularity) can be mechanically translated to one using only (Circularity), which is fully supported by our implementation.

9.5 EVALUATION

We evaluated our proof generation method using two benchmark sets. The first benchmark set consists of some verification problems of programs written in three programming languages, which aims at showing that our method is indeed language-agnostic. The second benchmark set is a selection of C verification examples from the SV-COMP competition [127]. We used a machine with Intel i7-12700K processors and 32 GB of RAM. The evaluation results are shown in Table 9.1. From left to right, we list the verification tasks, **spec**ification **LOC**, number of symbolic execution **steps**, proof **hint size**, **proof** object **size** (uncompressed/compressed), $\mathbb{K}$ verifier time (without proof generation), proof **gen**eration time, and proof **check**ing time (check 1 using `smetamath` [128] and check 2 using our own implementation in Rust [119]). Tasks with prefix `ln/` are from the `loop-new` benchmark of SV-COMP [127]. In the following, we discuss the benchmark sets and the evaluation results in detail.

To demonstrate that our proof generation method is language-agnostic, we defined three different programming languages in $\mathbb{K}$:

- IMP (see Figure 2.13): a simple imperative language with C-like syntax;
- REG: an assembly language for a register-based virtual machine;
- PCF, i.e., programming computable functions [129]: a typed functional language with a fixed-point operator.

We considered the following verification examples:

- SUM, which computes $1 + \cdots + n$ for input n;
- `EXP`, which computes n^k for inputs n and k;
- `COLLATZ`, which computes the Collatz sequence [130] for input n until it reaches 1;
- `PRODUCT`, which computes the product of integers using a loop.
- `GCD`, which computes the greatest common divisor of two integers using the Euclidean algorithm.

All benchmark programs and their formal specifications are implemented/specified in the three programming languages IMP, REG, and PCF. Table 9.1 shows that our prototype can generate proof objects for all these programs without additional effort. Besides these verification examples, we also considered the C programs from the `loop-new` benchmark set in the SV-COMP competition [127].

Even for simple arithmetic programs such as SUM, the symbolic execution process is complicated, as one can see from the proof object sizes in Table 9.1. A lot of seemingly innocuous operations that are performed by the $\mathbb{K}$ deductive verifier, such as substitution and equational simplification, result in very long proof objects, which encode proof steps down to the lowest possible level—the proof system.

We measured the performance of both proof generation and proof checking. For proof generation, we measured the generation time, the number of symbolic execution steps, the sizes of the proof hint and the final proof objects. We also measured the sizes of compressed proof objects using a generic compression tool `xz` [121]; these compressed proofs can be decompressed and checked on-the-fly using an online Metamath verifier such as `mmverify` [131].

At a high level, the proof generation time consists of (1) the time to generate the AML theory Γ^L from the $\mathbb{K}$ formal language semantics of L, and (2) the time to generate the proof objects using the procedures described in Chapter 9. In our experiments, (1) only takes a few seconds and is linear to the number of semantic rules. Most time is spent on (2), which is linear to the number of symbolic execution steps conducted during verification and the sizes of the intermediate configurations. Generally speaking, deductive verifiers are slow, and it takes even more time for users to propose the right invariants. In our view, it is therefore acceptable to spend the extra time on generating rigorous and machine-checkable proof objects for deductive verifiers and their verification runs, which help establish the correctness of the verification results on a smaller trust base.

Due to the simplicity of Metamath and the 240-line formalization of AML, it is very fast to check proof objects. Once the proofs are generated, they can be made public as machine-checkable correctness certificates of the verification tasks. Anyone concerning about the correctness of the verification can access the public proof objects, set up a proof checking environment (which is much simpler than setting up a verification environment), and check the proofs independently. We are optimistic about the scalability of our method on large $\mathbb{K}$ developments because proof checking scales well. The sizes of proof objects are linear to the number of symbolic execution steps and the sizes of configurations. The complexity of proof checking is also linear to the sizes of proof objects. We do not see a nonlinear factor or an exponential explosion in our proof generation method.

Metamath has its own format to compress proofs (see [107, Appendix B]). On top of that, proof objects can be compressed as plain text files using any mainstream compression tool such as `xz` [121], which leads to >95% reduction in the proof sizes, as shown in Table 9.1, at the expense of spending more time in decompressing the proofs for proof checking and using an online proof checker, which can be slower than an offline one. It is left as future work to study such space-time trade-off in proof checking and find the right balance.

9.6 DISCUSSION

We first discuss the trust bases of proof checking and $\mathbb{K}$ and then provide some interesting details about our prototype implementation.

9.6.1 Trust base of proof checking

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

Most existing works use a proof assistant such as Coq [116] or Isabelle [117] to encode and check the final proof objects. While proof assistants are commonly used in specifying and reasoning about computer systems, they are complex artifacts. For example, Coq has 200,000 lines of OCaml, and the safety-critical kernel still has 18,000 lines [118]. It means that if Coq is used as the final proof checker, there is at least 18,000 lines of OCaml code to be trusted. It is difficult for us to find the statistics for other proof assistants and/or theorem provers but we expect they are similar.

Metamath [107], on the other hand, is a tiny language that can express theorems in abstract mathematics, accompanied by proofs that can be checked by a program, called a Metamath verifier. Internally, the Metamath verifier behaves like an automaton with a stack. Axioms and theorems are associated with unique labels and a proof is a sequence of such labels. To check a proof, one maintains a stack that is empty initially, scans the proof, and pushes/pops the axioms and/or the hypotheses/conclusions of theorems accordingly. If in the end the stack contains exactly one statement that is identical to the theorem being proved, the proof is checked. In particular, it does not need to do any complex inference such as pattern matching or unification, making proof checking very simple. As a result, Metamath has dozens of independently-developed verifiers. [107] lists 19 of them, some of which are very small: 550 lines of C#, 400 lines of Haskell, 380 lines of Lua, and 350 lines of Python. As a proof-of-concept, we also implemented a Metamath verifier in 740 lines of Rust [119], which supports both regular and compressed proofs, and used it in our experiments.

In our work, we use Metamath to encode the proof objects. Also, we build on an existing formalization of AML and its proof system in 240 lines of Metamath code [113]. As for what counts as the actual proof checker in our approach, there can be different opinions, depending on whether Metamath is regarded as a programming language, or as another calculus whose inference system is implemented in a mainstream language, on top of which the proof system of AML is formalized. If Metamath is considered as a programming language, our proof checker has 240 lines. Otherwise, our proof checker consists of the 240-line Metamath definition plus an implementation of Metamath (550 lines of C#, 400 lines of Haskell, etc.), which in total has fewer than 1000 lines.

In our (maybe biased) view, there is no reason to *not* regard Metamath as a programming language like C# and Haskell. Metamath is much simpler than (almost) all programming languages. The fact that Metamath has many independent implementations using different programming languages makes it depend *less* on any particular programming language and its runtime environment, such as compilers and underlying operating systems. Metamath is also bootstrapping, in the sense that the executable of its own verifier (as a piece of machine code run on x86-64 Linux) is formally defined in Metamath itself [120, Section 6]. What is the highest possible correctness guarantee that we can expect from a proof checker? [120] proposes five possible levels to which we can prove the correctness of the checker, from the level of a logical rendering of the code to that of the logic gates that make up the computer and even the fabrication process relative to some electrical or physical model (although one may not want to do so because the result will be too specific to that particular computer or digital setup). It is clearly out of the scope of this paper to address all the above questions. The meta-point we want to make here is that proof checking systems such as Metamath have perhaps not received the attention they deserve from the formal verification and theorem proving community.

9.6.2 Trust base of $\mathbb{K}$

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

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

All the above frontend shortcuts and notations will be eliminated by the frontend of 𝕂. The frontend tool `kompile` translates the formal language semantics into an intermediate formal language called Kore [122], which is used to specify patterns and axioms. `kompile` parses all the concrete syntax into abstract syntax trees, represented as patterns. It also infers the omitted parts of configurations in semantic rules and the types of all the variables. In the end, `kompile` produces one Kore definition— as one source file `definition.kore`—that includes the entire AML encoding of the formal language semantics. The compiled Kore file is then passed to 𝕂's back-ends to generate the corresponding language tools.

Therefore, Kore behaves as the intermediate interface between the frontend and the back-ends. It is also the boundary between the informal and formal worlds. Since Kore is a formal specification language for writing AML theories, the formal semantics of a Kore definition *is* the AML theory that it defines. However, the frontend syntax of 𝕂 (as shown in Figure 2.13) does not (yet) have a formal semantics. Its meaning is completely determined by `kompile`, which lacks a formal specification.

In this work, we are interested in certifying back-end correctness. More precisely, we are certifying the language-agnostic deductive verifier, implemented by the Haskell back-end. Previously, the correctness of formal verification in 𝕂 depends on the 120,000-line Haskell back-end and its internal verification algorithm (Algorithm 9.1) as well as optimized, complex algorithms for symbolic execution and pattern matching/subsumption. By generating proof objects for these algorithms, we eliminate them from the trust base.

We should also clarity that the entire trust base for end-to-end verification in 𝕂 is still large and should be further reduced in the future. Firstly, the `kompile` tool belongs to the trust base. Secondly, the automatic encoder (developed in [114]) that translates Kore into Metamath belongs to the trust base (Figure 8.2), although the translation is very simple; it only parses the Kore definition and prints it in the Metamath format. Thirdly, the formalization of AML in Metamath belongs to the trust base, which is very small (240 lines).

However, all the back-end algorithms are no longer in the trust base. They are certified by AML proofs and the proof checker.

9.6.3 Implementation

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

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

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

We also developed a library of 196 lemmas about the rewriting and reachability relations such as Lemma 9.2 in Chapter 9. These lemmas were proved manually in Metamath in ∼4,000 lines and have been added to the existing Metamath database of AML. Note that all these lemmas are checked by the Metamath verifiers so they do not belong to the trust base.

We implemented several optimizations for constructing proof objects to improve performance. To avoid reproducing a (sub)-proof over and over again, we cache an incomplete work-in-progress proof when its size exceeds a certain threshold and add it as a lemma, which can be used in future proofs to reduce duplicates. To save runtime memory, we represent proof trees as directed acyclic graphs (DAGs) where the common subtrees are shared. When we apply an intermediate lemma or combine multiple DAGs, we use a greedy algorithm to merge the subtrees that have the same conclusion. Even with these optimizations, proofs are still huge (in the order of tens of megabytes), which is primarily due to the space-inefficient text-based encoding. To reduce the proof sizes further, we can compress the proofs using a generic compression tool such as `xz` [121], which provides >95% reduction in size; see Section 9.5 for more details.

The $\mathbb{K}$ deductive verifier consists of a powerful symbolic execution tool that supports many complex features such as evaluation order, conditional rewriting, "otherwise" rules (which are catch-all rules if no other semantic rules can be applied), user-defined contexts, unification modulo axioms, etc. Our current prototype implementation supports proof generation for a significant subset of these features. For evaluation orders, $\mathbb{K}$ specifies them using strictness attributes (Section 2.15), which are reduced to a special case of conditional rewriting, which is supported by our tool. The "otherwise" rules are also reduced to conditional rewriting where the condition states that no other semantic rules are applicable, and thus are also supported by our tool. $\mathbb{K}$ also provides a more advanced (but also much less often used) way to define evaluation orders using explicit user-defined contexts, which is not supported by our tool yet. Finally, unification modulo maps (i.e., unification modulo associativity, commutativity, and units) is supported. Currently, the logical encoding of a $\mathbb{K}$ semantics is computed by a frontend tool called `kompile` (see Figure 8.2), which lacks a clear documentation of the axioms it generates. This makes developing the proof generation procedure harder because we need to manually find suitable classes of axioms in `kompile`'s output. Therefore, we expect supporting proof generation for large real-world $\mathbb{K}$ developments to be a long-term endeavor, which involves a formalization of `kompile` and requires a close collaboration with the $\mathbb{K}$ team (see Section 9.6.2 for more discussion on `kompile`).

9.6.4 Future directions

We identify some main future directions of the current work. Firstly, as discussed in Section 9.6.2, the frontend tool `kompile` needs to be trusted. It is not satisfying, because the frontend consists of roughly 40,000 lines of Java, while many tasks that it performs, such as configuration inference and completion, can also be formalized as AML proofs, the same way how program execution and deductive verification are AML proofs. In the long run, we see no reason to not formalize the *entire* $\mathbb{K}$ frontend, even including the parser. Indeed, the concrete syntax given by a context-free grammar can be regarded as the initial algebra of an equational/algebraic specification [123]. A parser can then be specified as a function from the domain of strings (sequences of characters) to that initial algebra. Since initial algebra semantics can be defined in AML [124], the parsing function can be inductively axiomatized and certified by AML proofs.

The second future direction is to incorporate proofs for SMT solvers. Currently, our implementation trusts SMT solvers and does not generate proof objects for them. $\mathbb{K}$ uses SMT solvers for domain reasoning, such as $\Gamma^L \vdash \varphi \to \psi$, where φ and ψ are logical constraints about domain values such as integers. To prove such domain properties, we encode them as equivalent FOL formulas and query an SMT solver, thus resulting in a gap in our proof objects that needs to be addressed separately in the future, following existing research such as [111, 125].

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

- The subsumption proof generation (Section 9.3) may not match the actual simplification procedure of the $\mathbb{K}$ verifier, thus resulting in subsumptions that are correctly done by $\mathbb{K}$ but cannot be proved by our proof generation tool.
- Our proof generation procedure does not support the (Set Circularity) rule as discussed in Section 9.4, while the $\mathbb{K}$ verifier does use (Set Circularity) in general.

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

Finally, as discussed in Section 9.1, we plan to extend our proof generation method to support proof generation for all-path reachability reasoning [3, 126]. In the current work, we only consider one-path reachability logic, which captures the partial correctness of one execution trace. For nondeterministic and concurrent programs, we need all-path reachability logic to prove the correctness of all execution traces. All-path reachability logic is proposed for precisely that purpose. An all-path reachability claim $\varphi \Rightarrow^{\forall}_{\mathit{reach}} \psi$ holds iff for every maximal and finite execution traces starting from φ, ψ is reachable. The proof system of all-path reachability logic has identical proof rules as one-path reachability logic in Figure 2.12 (replacing $\Rightarrow$ with $\Rightarrow^{\forall}_{\mathit{reach}}$), except one additional axiom called (Step)

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

where $A = \{\mathit{lhs}_1 \Rightarrow \mathit{rhs}_1, \ldots, \mathit{lhs}_K \Rightarrow \mathit{rhs}_K\}$ is the set of all the semantic rules, which are one-path rules in nature. The (Step) axiom derives all-path claims from these semantic rules, where ψ_k is the result of executing φ for one step, using the k-th semantic rule $\mathit{lhs}_k \Rightarrow \mathit{rhs}_k$ for $1 \leq k \leq K$. Thus, the (Step) axiom states that the only way to make an execution step is to use one of the semantic rules in A. Since the current $\mathbb{K}$ pipeline that translates $\mathbb{K}$ into AML (Figure 8.2) is incomplete and the resulting theory Γ^L does not have the (Step) axiom, proof generation for all-path reachability claims is left as future work.

Chapter 10: RELATED WORK

We present related work and compare them with this work on the following topics: (1) existing approaches to programming language frameworks; (2) existing approaches to defining binders; (3) existing approaches to automated fixpoint reasoning; and (4) existing approaches to trustworthy programming language tools.

10.1 FORMAL SEMANTICS AND PROGRAMMING LANGUAGE FRAMEWORKS

It is hard to discuss, even summarily, the over half a century of research in formal semantics and programming language frameworks. Since the 1960s, various semantics notions and styles have been proposed and become canonical approaches to defining formal semantics, including Floyd-Hoare axiomatic semantics [12, 13], Scott-Strachey denotational semantics [14], initial algebra semantics [123], and various types of operational semantics [15, 16, 17, 18]. A nice survey about the earlier research in formal semantics and semantic frameworks in the past centenary can be found in [132]. More recent work will be discussed shortly after. By collecting these references to related work, we realize how much progress we have been made since the first paper on formal semantics of programs published in 1960s, and how close we are to reaching the ideal language framework vision (Figure 1.1).

CENTAUR [133] is one of the earliest attempts in developing a system that takes formal language definitions and automatically generates programming environments, which consist of many language tools, including interpreters and debuggers, equipped with graphic interfaces.

Proof assistants such as Coq [116] and Isabelle [117] represent an important trend to define the formal semantics of programming languages. Programs and program configurations are defined as data structures, and various types of formal semantics can be defined as functions or relations on these data structures. Program execution and verification can be done in a manual, semi-automatic, or fully automatic manner, with or without human interference. Meta-properties, such as the equivalence between the two different semantics of a language, can be proved, but often require remarkably effort. Formal syntax is often not considered.

Due to the complexity of aforementioned proof assistants, lightweight tools such as Ott [134] occurred, serving as an expressive and intuitive front-end to write formal syntax and semantics definitions of programming languages and calculi. Automatic tools such as those which sanity-check the formal definitions or translate definitions to proof assistants, are implemented.

Component-based specification (CBS) framework [135] observes that many programming languages share a variety of many fundamental programming constructs, or simply *foncons*.

CBS framework allows one to define the formal semantics of programming languages by translating them to foncons in a component-based and modular way, aiming at good reusability of formal definitions.

Spoofax [136] is a platform for designing programming languages, in particular domain specific languages (DSL), with an integration of language tools, including syntax definition formalism such as SDF [137], program translation and code generation tools such as Stratego [138], program analysis tools such as data flow analyzer FlowSpec [139].

PLT Redex [140], which is now embedded in the programming language Racket, is a DSL for designing formal syntax and operational semantics as reduction rules. Random programs can be automatically generated that serve as tests of the semantics.

Rosette [141] is a solver-aided programming language that extends Racket with a small set of language constructs for program verification and synthesis. Language designers, often of DSL, implement interpreters in Rosette, and by symbolic evaluation, the language synthesis and verification tools are generated for free. Racket has helped non-expert users to design and create solver-aided tools for various domains. Research on symbolic profiling examines process of symbolic evaluation and proposes techniques that automatically fix performance bottlenecks of the generated tools [142].

10.2 EXISTING APPROACHES TO DEFINING BINDERS

We discuss some existing approaches to defining binders and compare them with our approach using matching μ-logic, as presented in Sections 5.12 and 5.13. These approaches include: (1) de Bruijn techniques [143], which give α-equivalent terms identical encodings; (2) combinators [28], which translate terms with binders to binder-free combinator terms; (3) nominal logic [144], which uses first-order logic (FOL) to axiomatize name-swapping and freshness, and uses them to axiomatize object-level binding; (4) higher-order abstract syntax [145] (abbreviated HOAS), which uses fixed binders in the meta-language, often a variant of typed λ-calculus, to define arbitrary binders in the object-level systems; (5) explicit substitution [146], which uses customized calculi where the meta-level operation of capture-avoiding substitution is incarnated in an object-level operation as part of the calculi; (6) term-generic logic [30] (abbreviated TGL), which is a FOL variant parametric in a generic term set, defined axiomatically and not constructively, which can be instantiated by a concrete binder syntax. We discuss how these approaches handle binders and binding behavior using the following λ-expression as an example (a closed expression with distinct bound variables, which requires α-renaming during reduction to avoid variable-capture):

$$(\lambda z\,.\,(zz))(\lambda x\,.\,\lambda y\,.\,(xy)) \tag{†}$$

De Bruijn encodings eliminate bound variables by replacing them with indexes that denote the number of (nested) binders that are in scope between them and their corresponding binders.[1] For example, the de Bruijn encoding of (†) is $(\lambda(11))(\lambda\lambda(21))$, where 1 means that it is bound by the closest binder and 2 means that it is bound by the second closest binder. Bound variables are eliminated so α-equivalent expressions have the same de Bruijn encoding. However, substitution requires index shifting, to adjust the indexes. De Bruijn techniques are used as the internal representations of terms in several theorem provers, but the encoding is not human readable, implementations are often tricky to get right, and efficiency problems can still appear on large terms.

Combinators translate binders to binder-free terms, which are built with constants like k and s, and application. This translation is called abstraction elimination, and can be implemented using term rewriting [147]. It may cause exponential growth in the translated term size. Reduction of combinatory terms is done using equations like $kxy = x$ and $sxyz = (xz)(yz)$ regarded as rewrite rules. Combinatory terms are not human readable; for example, (one of) the equivalent combinator term of (†) is $s(skk)(skk)s(s(ks)(s(kk)(skk)))(k(skk))$. Using combinators, the binding behavior of λ is captured *implicitly* through abstraction elimination.

Nominal logic refers to a family of FOL theories whose signatures contain a name-swapping operation $(x\,y)\cdot e$ that swaps all (free and bound) occurrences of x and y in e, and a freshness predicate $x \# e$ stating that x has no free occurrences in e. The notions of α-equivalence and capture-avoiding substitution are then axiomatized using additional FOL axioms on top of the axioms of name-swapping and freshness. As an example, the following is an axiom in [144, Appendix A.3] that states that swapping two fresh names that do not occur free in a term has not effect:

$$\text{(F1)}\quad \forall x\,{:}\,V\,.\,\forall y\,{:}\,V\,.\,\forall e\,{:}\,\mathsf{Exp}\,.\,x \# e \wedge y \# e \to (x\,y)\cdot e = e$$

where V and Exp are the sorts of variables (also called atoms) and expressions, respectively. Nominal logic also defines a new sort $[V]\mathsf{Exp}$ and a FOL binary function $\_.\_ : V \times \mathsf{Exp} \to [V]\mathsf{Exp}$ for binding, whose properties such as α-equivalence are axiomatized. Then, β-reduction

[1] Other de Bruijn encodings count the binders from the top of the terms.

in λ-calculus, e.g., can be defined in the following way [148, pp. 251, Eq. (12.17)]:

$$(\beta \text{ IN NOMINAL LOGIC}) \qquad \forall x : V \,.\, \forall e : \mathsf{Exp} \,.\, \forall e' : \mathsf{Exp} \,.\, \mathit{app}(\mathit{lam}(x.e), e') = \mathit{subst}((x.e), e')$$

where $\mathit{subst}(\_,\_)$ is a binary function defined by four axioms (see [144, pp. 8]), in accordance to the four possible forms that e can take (i.e., the variable x; a variable distinct from x; application; or abstraction). E.g., the following is the substitution axiom for abstraction [148, Eq. (12.20)]:

$$\forall x : V \,.\, \forall y : V \,.\, \forall e : \mathsf{Exp} \,.\, \forall e' : \mathsf{Exp} \,.\, y \# e' \to \mathit{subst}(x \,.\, \mathit{lam}(y \,.\, e), e') = \mathit{lam}(y \,.\, \mathit{subst}(x \,.\, e, e'))$$

Note that x and e are meta-variables in λ-calculus and become normal variables in nominal logic, so the whole embedding is a deep embedding. (Compare this to our encoding in **??** where meta-variables x and e in λ-calculus are still meta-variables in matching μ-logic.)

Besides nominal logic and its metatheory [149, 150, 151], there is a wider range of research on nominal techniques in general, including studies on using Fraenkel-Mostowski sets [152], nominal sets [153] or similar set-theoretic structures [154] as well as category-theoretic notions [155] to formalize and reason about binders and operations on them, and have resulted in practical implementations that support complex recursive and inductive reasoning over terms with bindings as well as algorithms for unification [156] and narrowing [157]. These nominal approaches deal with variable names and bindings directly, treat variable names as normal data that can be manipulated, quantified, and reasoned about, and give explicit definitions to operations such as free variables and capture-avoiding substitution (via name-swapping and freshness). Note that nominal approaches can be directly exploited in matching μ-logic because FOL is a methodological fragment of matching μ-logic.

Higher-order abstract syntax (HOAS) is a design pattern where some expressive higher-order calculus, usually one of the variants of typed λ-calculus [145, 158, 159, 160, 161, 162] or second-order equational logic [161, 163], is used as a foundation to define object-level binders. As an example, we show (part of) the HOAS-style definition of (untyped) λ-calculus in the Twelf system [164]:

```
exp : type.                                     // the type of λ-expressions
app : exp -> exp -> exp.                        // function application
lam : (exp -> exp) -> exp.                      // function abstraction
red : exp -> exp -> type.                       // reduction relation
red-beta : red (app (lam ([x] (F x))) E) (F E). // β-reduction
```

where in red-beta, [x] _ is the built-in binder of (the HOAS variant underlying) Twelf; E is a variable of type exp; F is a variable of the function type exp -> exp; and (F x) is the (metalevel) application of F to x. Higher-order matching is needed when red-beta is applied, and the internal substitution mechanism of Twelf is triggered when F is applied to E. The binding behavior of λ is obtained from the binding behavior of the built-in binder [x] _, via a constant lam; specifically, $\lambda x.e$ is encoded as lam ([x] e). Object-level substitution is avoided, but clearly this is not how β-reduction is usually defined (for the usual definition, see (β, Reduction) below). Application in λ-calculus is defined by a simple desugaring to the builtin application, using a different constant app; that is, $e_1\, e_2$ is defined as app $e_1\, e_2$ (rather than $e_1\, e_2$). Thus, the definition needs to be justified by proving *adequacy theorems* that establish a bijection between the expressions and formal proofs of λ-calculus, and the HOAS terms and type derivations, which is a tedious and nontrivial task [165].

Explicit substitution turns the implicit meta-level substitution operation into more explicit and atomic steps, in order to provide a better understanding of the operational semantics and execution models of higher-order calculi (see [166, pp. 1–2]; see also [167, pp. 4] for historical remarks). By doing so, it bridges the gap between higher-order formalisms and their implementations, and has resulted in several practical tools. For example, [168] proposes a calculus for explicit substitution whose implementation allows us to define executable formal representations of many logical systems featuring binders with a close-to-zero representational distance.

Term-generic logic (TGL), as we have seen in Section 2.12, is a FOL variant, where the set of terms T is generic and given as a *parameter* that exports two operations—free variables and capture-avoiding substitution—satisfying certain properties [30, Definition 2.1]. TGL formulas are then defined constructively as in FOL, from predicates $\pi(e_1, \dots, e_n)$ and equations $e_1 = e_2$, to compound formulas built using $\wedge$, $\neg$, and $\exists$, with the important exception that $e_1, \dots, e_n$ are not constructive terms like in FOL, but generic terms in T. In the case of λ-calculus, the set of λ-expressions Λ can be proved to satisfy the definition of a generic term set in TGL, so we can *instantiate* TGL by Λ. The binding behavior of λ is inherited automatically, through the T instance. The metalevel of λ-calculus can be defined by TGL axioms. For example, β-reduction is captured either as an equation or as a relation:

$$(\beta,\ \textsc{Equation}) \quad (\lambda x\,.\,e)\,e' = e'[e/x] \qquad\qquad (\beta,\ \textsc{Reduction}) \quad \mathit{reduces}\big((\lambda x\,.\,e)\,e', e'[e/x]\big)$$

where *reduces* is a binary predicate; $(\lambda x\,.\,e)\,e', e'[e/x] \in \Lambda$ are generic terms (schemas) that represent all the concrete instances. TGL has been used to define various systems featuring bindings. In this paper, we use TGL as an intermediate to capture other systems with binders within matching μ-logic.

We have shown that matching μ-logic provides a general approach to defining binders in Sections 5.12 and 5.13. An important aspect of the matching μ-logic-based approach is that it yields *models*. Models are insightful. They help us understand a logical system better, from a different angle. It is not unusual that more than one notion or class of models are proposed for one logic, because each has its unique merit in helping us understand the logic from a certain perspective. Since matching μ-logic has a built-in notion of models, by defining a logical system as a matching μ-logic theory we can immediately study its resulting model theory and properties. For example, the matching μ-logic theory of λ-calculus yields a precise and insightful description of how $\lambda x \,.\, e$ is interpreted (semantically) in matching μ-logic models, which leads us to a new semantics of λ-calculus that is representationally complete for all λ-theories (Section 5.12.4).

10.3 EXISTING APPROACHES TO AUTOMATED FIXPOINT REASONING

Here we discuss other approaches to automated fixpoint reasoning and compare them with our unified proof framework from a methodology point of view.

We were inspired and challenged by work on automation of inductive proofs for separation logic [22], which resulted in several automatic separation logic provers; see [96] for those that participated in the recent SL-COMP'19 competition. Since separation logic is undecidable [169], many provers implement only decision procedures to decidable fragments [23, 91, 92, 102] or incomplete algorithms [88, 89, 90]. There is also work on decision procedures for other heap logics [170, 171, 172, 173, 174, 175], which achieve full automation but suffer from lack of expressiveness and generality. It is worth noting that significant performance improvements can be obtained by incorporating first-order theorem proving and SMT solvers [97, 98] into separation logic provers [176, 177].

Compared with our unified proof framework, the above provers are specialized to separation logic reasoning. Some are based on reductions from separation logic formulas to certain decidable computational domains, such as the satisfiability problem for monadic second-order logic on graphs with bounded tree width [89]. Others are based on separation logic proof trees, where the syntax of separation logic has been hardwired in the prover. For example, most separation logic provers require the following canonical form of separation logic formulas: $\varphi_1 * \cdots * \varphi_n \wedge \psi$ where $\varphi_1, \ldots, \varphi_n$ are basic spacial formulas built from singleton heaps $x \mapsto y$ or user-defined recursive structures such as $\mathit{list}(x)$, and ψ is a FOL logical constraint. This built-in separation logic syntax limits the use of these provers to separation logic, even though the inductive proof rules proposed by the above provers might be more general. The major advantage of our unified proof framework, which was the motivation fueling our effort, is that the inductive principle can be applied to any structures, not only those representing heap structures. In Section 6.1.1, we show the key elements of our proof framework that supports the fixpoint reasoning for arbitrary structures.

Hoare-style formal verification represents another important but specialized approach to fixpoint reasoning, where the objects of study are program executions and the properties to prove are program correctness claims. There is a vast literature on verification tools based on classical logics and SMT solvers such as Dafny [178], VCC [179] and Verifast [180]. To use these tools, the users often need to provide annotations that explicitly express and manipulate frames, whose proofs are based on user-provided lemmas. The correctness of the lemmas is either taken for granted or manually proved using an interactive proof assistant (e.g., [179, Section 6] mentions several tools that are based on Coq [116] or Isabelle [117]). While it is acceptable for deductive verifiers to take additional annotations and/or program invariants, the use of manually-proved lemmas is not ideal because it makes the verification tools not fully automatic.

An interesting approach to formal verification that inspired this paper is reachability logic [3], which uses the operational semantics of a programming language to verify the programs of that language, using one fixed proof system. In that sense, it shares a similar vision with our unified proof framework, where the formal semantics of programming languages are defined as the logical theories and only one proof system is needed to verify all programs written in all languages. In Sections 2.14, we will show how our proof framework can carry out reachability-style formal reasoning, and thus support program verification in a unified way.

There is recent work that considers inductive reasoning for more general data structures, beyond only heap structures [103, 181, 182, 183, 184, 185]. Tac [186] is an automated theorem prover for a variant of FOL extended with fixpoints that uses the techniques of *focusing* to reduce the nondeterminism involved in proof search. [103] proposes CYCLIST, a proof framework that implements a generic notion of *cyclic proof* as a “design pattern” about how to do inductive reasoning, which generalize the proof systems of LFP and SL. In CYCLIST, inductive reasoning is achieved not by an explicit induction proof rule, but implicitly by cyclic proof trees with “back-links”. In contrast, our unified proof framework uses one fixed logic (matching μ-logic) and relies on an explicit induction proof rule (KNASTER TARSKI). Therefore, CYCLIST represents a different approach from ours but towards a similar goal of a unified framework for fixpoint reasoning.

10.4 EXISTING APPROACHES TO TRUSTWORTHY LANGUAGE TOOLS

There has been a lot of effort in providing formal guarantees for programming language tools such as compilers or deductive verifiers. At a high level, we may identify two approaches. One approach is to formalize and prove the correctness of the entire tool. For example, CompCert C [187] is a C compiler that has been formally verified to be exempt from miscompilation issues. The other approach is to generate proof objects on a case-by-case basis for *each run* of the tool. For example, [188] presents the translation validation technique to check the result of each compilation against the source program and [189] presents an approach where successful runs of the Boogie verifier are validated using Isabelle proofs. Our work belongs to the second approach, where proof objects are generated for each verification task carried out using $\mathbb{K}$.

The first approach tends to yield proofs that are more technically involved and does not work well on an existing tool implementation, and is often conducted on a new implementation that aims at being correct-by-construction from the beginning. However, once it is done, it gives the highest formal guarantee for the correctness of the entire tool, once and for all. Besides CompCert C that we mentioned above, there is also CakeML [190], which is an implementation of Standard ML [191] that is formally verified in HOL4 [192]. In this approach, the proof objects are often written and proved in an interactive theorem prover such as Coq [116] and Isabelle [117], because they provide the expressive power needed to state the correctness claims, which are often higher-order, in the sense that they are quantified over all possible programs and/or inputs.

The other "case-by-case" approach generates simpler proof objects and works better on an existing tool implementation, compared to the above "once-and-for-all" approach. In this approach, the proof objects only relate the input and output of the language tool in question, without needing to depend on the actual implementation of the tool. For example, the technique of translation validation [188] checks the correctness of each compilation of an optimized compiler, producing a *verifying* compiler, in contrast to a *verified* compiler such as CompCert C. Recently, the idea was applied to not only compilers but also interpreters and deductive verifiers. For example, [114] generates proof objects for a language-agnostic interpreter, where each (concrete) execution of a program is certified by a machine-checkable mathematical proof. [189] generates proof objects for the intermediate verification language (IVL) Boogie, where each transformation from programs to their verification conditions is certified. [193] generates proof objects for the Why3 verifier [194], which is also equipped with an IVL to generate verification conditions. [195] generates proof objects for the VeriFast verifier for C [196], where each successful verification run is certified with respect to CompCert's Clight big step semantics [197]. There have also been works that generate proofs for the decision procedures in SMT solvers to certify their correctness [111, 125, 198].

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

Chapter 11: CONCLUSION

We have presented matching μ-logic, which is a unifying logic for specifying and reasoning about programs and programming languages. We have studied the proof theory and expressive power of matching μ-logic and proved the soundness theorem and a few important completeness results. We have studied automated fixpoint reasoning and proposed a set of high-level automated proof rules for matching μ-logic. We have proposed applicative matching μ-logic (AML) as a simple instance of matching μ-logic that remains all of its expressive power, and implemented a proof checker for AML. Finally, we have studied study proof-certifying program execution and formal verification by implementing proof generation procedures for a language-independent interpreter and a language-independent formal verifier of the $\mathbb{K}$ framework. This way, the correctness of program execution or formal verification is reduced to checking the corresponding AML proof objects using the 240-line proof checker.

We hope to demonstrate the feasibility of having matching μ-logic serve as a unifying foundation for programming, where programming languages are defined as logical theories, and the correctness of language implementations and tools is established by logical proof objects, which can be checked by a small proof checker.

References

[1] "K Framework Tools," https://github.com/runtimeverification/k, 2023.

[2] G. Roşu, "Matching logic," *Logical Methods in Computer Science*, vol. 13, no. 4, pp. 1–61, 2017.

[3] A. Ştefănescu, D. Park, S. Yuwen, Y. Li, and G. Roşu, "Semantics-based program verifiers for all languages," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16).* ACM, 2016, pp. 74–91.

[4] G. Rosu, "K—A semantic framework for programming languages and formal analysis tools," in *Dependable Software Systems Engineering.* IOS Press, 2017.

[5] C. Hathhorn, C. Ellison, and G. Roşu, "Defining the undefinedness of C," in *Proceedings of the 36th annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15).* Portland, OR: ACM, 2015, pp. 336–345.

[6] D. Bogdănaş and G. Roşu, "K-Java: A complete semantics of Java," in *Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL'15).* Mumbai, India: ACM, 2015, pp. 445–456.

[7] D. Park, A. Ştefănescu, and G. 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).* Portland, OR: ACM, 2015, pp. 346–356.

[8] D. Guth, "A formal semantics of Python 3.3," M.S. thesis, University of Illinois at Urbana-Champaign, Aug. 2013. [Online]. Available: http://hdl.handle.net/2142/45275

[9] E. Hildenbrandt, M. Saxena, X. Zhu, N. Rodrigues, P. Daian, D. Guth, B. Moore, Y. Zhang, D. Park, A. Ştefănescu, and G. Roşu, "KEVM: A complete semantics of the Ethereum virtual machine," in *Proceedings of the 2018 IEEE Computer Security Foundations Symposium (CSF'18).* Oxford, UK: IEEE, 2018, http://jellopaper.org. pp. 204–217.

[10] S. Dasgupta, D. Park, T. Kasampalis, V. S. Adve, and G. 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).* Phoenix, Arizona, USA: ACM, 2019, pp. 1133–1148.

[11] G. Roşu, A. Ştefănescu, Ş. Ciobâcă, and B. M. Moore, "One-path reachability logic," in *Proceedings of the 28th Symposium on Logic in Computer Science (LICS'13).* IEEE, 2013, pp. 358–367.

[12] V. Pratt, “Semantical consideration on Floyd-Hoare logic,” in *Proceedings of the 17th Annual Symposium on Foundations of Computer Science (SFCS’76)*. IEEE, 1976, pp. 109–121.

[13] C. A. R. Hoare, “An axiomatic basis for computer programming,” *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[14] D. Scott, “Domains for denotational semantics,” in *International Colloquium on Automata, Languages, and Programming*, Springer. Berlin Heidelberg, Germany: Springer, 1982, pp. 577–610.

[15] G. D. Plotkin, “A structural approach to operational semantics,” *Journal of Logic & Algebraic Programming*, vol. 60-61, pp. 17–139, 2004.

[16] G. Kahn, “Natural semantics,” in *Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science (STACS’87)*, vol. 247, Passau, Germany, 1987, pp. 22–39.

[17] P. D. Mosses, “Modular structural operational semantics,” *Journal of Logic & Algebraic Programming*, vol. 60-61, pp. 195–228, 2004.

[18] G. Berry and G. Boudol, “The chemical abstract machine,” *Theoretical Computer Science*, vol. 96, no. 1, pp. 217–248, 1992.

[19] J. A. Goguen and J. Meseguer, “Completeness of many-sorted equational logic,” *Houston Journal of Mathematics*, vol. 11, no. 3, pp. 307–334, 1985.

[20] J. Meseguer and J. A. Goguen, “Initiality, induction, and computability,” in *Algebraic Methods in Semantics*. New York, USA: Cambridge University Press, 1985, pp. 459–543.

[21] R. M. Burstall and J. A. Goguen, *Algebras, theories and freeness: an introduction for computer scientists*, ser. NATO Advanced Study Institutes Series (Series C — Mathematical and Physical Sciences). Dordrecht, Netherlands: Springer, 1982, vol. 91, ch. 11, pp. 329–349. [Online]. Available: https://doi.org/10.1007/978-94-009-7893-5_11

[22] J. 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)*. Copenhagen, Denmark: IEEE, 2002, pp. 55–74.

[23] J. Brotherston, C. Fuhs, J. A. N. Pérez, and N. Gorogiannis, “A decision procedure for satisfiability in separation logic with inductive predicates,” in *Proceedings of the Joint Meeting of the 23rd EACSL Annual Conference on Computer Science Logic (CSL’14) and the 29th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS’14)*, ser. CSL-LICS ’14. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2603088.2603091 pp. 25:1–25:10.

[24] D. Kozen, “Results on the propositional μ-calculus,” *Theoretical Computer Science*, vol. 27, no. 3, pp. 333–354, 1983.

[25] M. J. Fischer and R. E. Ladner, “Propositional dynamic logic of regular programs,” *Journal of Computer and System Sciences*, vol. 18, no. 2, pp. 194–211, 1979.

[26] D. Harel, “Dynamic logic,” in *Handbook of Philosophical Logic.* Springer, 1984, vol. 165, pp. 497–604.

[27] D. Harel, J. Tiuryn, and D. Kozen, *Dynamic logic.* MIT Press, 2000.

[28] A. Church, *The calculi of lambda-conversion.* Princeton, New Jersey, USA: Princeton University Press, 1941.

[29] H. Barendregt, *The lambda calculus, its syntax and semantics*, ser. Studies in Logic. King’s College London, Strand, London WC2R 2LS, UK: College Publications, 1984.

[30] A. Popescu and G. Roşu, “Term-generic logic,” *Theoretical Computer Science*, vol. 577, pp. 1–24, 2015.

[31] F. Lucio-Carrasco and A. Gavilanes-Franco, “A first order logic for partial functions,” in *Proceedings of the 6th Annual Symposium on Theoretical Aspects of Computer Science (STACS’89).* Paderborn, Germany: Springer, 1989, pp. 47–58.

[32] M. Clavel, F. Durán, S. Eker, S. Escobar, P. Lincoln, N. Martí-Oliet, J. Meseguer, R. Rubio, and C. Talcott, *Maude manual (version 3.0)*, SRI International, 2020. [Online]. Available: http://maude.lcc.uma.es/maude30-manual-html/maude-manual.html

[33] J. R. Shoenfield, *Mathematical logic.* Addison-Wesley Pub. Co, 1967.

[34] P. Blackburn, M. d. Rijke, and Y. Venema, *Modal logic.* One Liberty Plaza, New York, NY: Cambridge University Press, 2001.

[35] I. Leustean and N. Moanga, “A many-sorted polyadic modal logic,” *CoRR*, vol. abs/1803.09709, 2018. [Online]. Available: http://arxiv.org/abs/1803.09709

[36] A. G. Hamilton, *Logic for mathematicians.* Cambridge, UK: Cambridge University Press, 1978.

[37] P. Blackburn and M. Tzakova, “Hybrid completeness,” *Logic Journal of IGPL*, vol. 6, no. 4, pp. 625–650, 1998.

[38] M. Schönfinkel, “Über die bausteine der mathematischen logik,” *Mathematische annalen*, vol. 92, no. 3-4, pp. 305–316, 1924.

[39] H. B. Curry, *Combinatory logic.* Amsterdam: North-Holland Pub. Co., 1958.

[40] A. Church, “A formulation of the simple theory of types,” *The Journal of Symbolic Logic*, vol. 5, no. 2, pp. 56–68, 1940.

[41] A. I. Malc’ev, “Axiomatizable classes of locally free algebras of various type,” *The Metamathematics of Algebraic Systems: Collected Papers*, vol. 1, no. 1, pp. 262–281, 1936.

[42] L. Kovács, S. Robillard, and A. Voronkov, "Coming to terms with quantified reasoning," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'17).* Paris, France: ACM, 2017, pp. 260–270.

[43] L. Löwenheim, "Über möglichkeiten im relativkalkül," *Mathematische Annalen*, vol. 76, no. 4, pp. 447–470, 1915.

[44] A. Kaposi, A. Kovács, and T. Altenkirch, "Constructing quotient inductive-inductive types," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019. [Online]. Available: https://doi.org/10.1145/3290315

[45] M. P. Fiore, A. M. Pitts, and S. C. Steenkamp, "Constructing infinitary quotient-inductive types," in *Proceedings of the 23rd International Conference on Foundations of Software Science and Computation Structures (FOSSACS'20) Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS'20)*, ser. Lecture Notes in Computer Science, J. Goubault-Larrecq and B. König, Eds., vol. 12077. Dublin, Ireland: Springer, 2020. [Online]. Available: https://doi.org/10.1007/978-3-030-45231-5_14 pp. 257–276.

[46] R. M. Burstall, "Proving properties of programs by structural induction," *The Computer Journal*, vol. 12, no. 1, pp. 41–48, 1969.

[47] J. McCarthy, "A basis for a mathematical theory of computation," in *Computer Programming and Formal Systems*, ser. Studies in Logic and the Foundations of Mathematics, P. Braffort and D. Hirschberg, Eds. Amsterdam, The Netherlands: Elsevier, 1963, vol. 35, pp. 33–70.

[48] D. C. Cooper, "The equivalence of certain computations," *The Computer Journal*, vol. 9, no. 1, pp. 45–52, May 1966.

[49] J. Mccarthy and J. Painter, "Correctness of a compiler for arithmetic expressions," in *Proceedings of Symposiain Applied Mathematics*, vol. 19. Rhode Island, USA: American Mathematical Society, 1967, pp. 33–41.

[50] R. M. Burstall, "Semantics of assignment," *Machine Intelligence*, vol. 2, pp. 3–20, 1968.

[51] J. A. Painter, "Semantic correctness of a compiler for an Algol-like language," *Stanford Artificial Intelligence Memo. No. 44*, vol. 1, no. 1, pp. 1–260, 1967.

[52] D. M. Kaplan, "Correctness of a compiler for Algol-like programs," *Stanford Artificial Intelligence Memo No. 48*, vol. 48, no. 1, pp. 1–35, 1967.

[53] H. Comon, "Inductionless induction," in *Handbook of automated reasoning*, A. Robinson and A. Voronkov, Eds. Amsterdam: North Holland, 2001, ch. 14, pp. 913–962.

[54] J. Meseguer, "Twenty years of rewriting logic," *The Journal of Logic and Algebraic Programming*, vol. 81, no. 7–8, pp. 721–781, 2012.

[55] J. Hendrix, J. Meseguer, and H. Ohsaki, “A sufficient completeness checker for linear order-sorted specifications modulo axioms,” in *Automated Reasoning*, U. Furbach and N. Shankar, Eds. Berlin, Heidelberg: Springer, 2006, pp. 151–155.

[56] J. Hendrix and J. Meseguer, “On the completeness of context-sensitive order-sorted specifications,” in *Term Rewriting and Applications*, F. Baader, Ed. Berlin, Heidelberg: Springer, 2007, pp. 229–245.

[57] C. Rocha and J. Meseguer, “Constructors, sufficient completeness, and deadlock freedom of rewrite theories,” in *Logic for Programming, Artificial Intelligence, and Reasoning*, C. G. Fermüller and A. Voronkov, Eds. Berlin, Heidelberg: Springer, 2010, pp. 594–609.

[58] J. D. Hendrix, “Decision procedures for equationally based reasoning,” Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2008.

[59] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi, “Tree automata techniques and applications,” Available on: http://www.grappa.univ-lille3.fr/tata, 2007, release October 12th, 2007.

[60] J.-P. Jouannaud and E. Kounalis, “Automatic proofs by induction in theories without constructors,” *Information and Computation*, vol. 82, no. 1, pp. 1–33, 1989.

[61] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud, *Software engineering with OBJ: Algebraic specification in action.* Massachusetts, USA: Springer, 2000, ch. Introducing OBJ, pp. 3–167.

[62] R. Diaconescu and K. Futatsugi, *CafeOBJ report: the language, proof techniques, and methodologies for object-oriented algebraic specification*, ser. AMAST Series in Computing. Singapore: World Scientific, 1998, vol. 6.

[63] A. Tarski, “A lattice-theoretical fixpoint theorem and its applications,” *Pacific Journal of Mathematics*, vol. 5, no. 2, pp. 285–309, 1955.

[64] I. Walukiewicz, “Completeness of Kozen’s axiomatisation of the propositional μ-calculus,” *Information and Computation*, vol. 157, no. 1-2, pp. 142–182, 2000.

[65] G. Lenzi, “The modal μ-calculus: A survey,” *Task quarterly*, vol. 9, no. 3, pp. 293–316, 2005.

[66] R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho, “Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems,” in *Hybrid systems.* Springer, 1993, pp. 209–229.

[67] E. A. Lee, “Cyber physical systems: Design challenges,” in *Proceedings of the 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing (ISORC’08).* IEEE, 2008, pp. 363–369.

[68] P. Blackburn, J. van Benthem, and F. Wolter, Eds., *Handbook of modal logic*, 1st ed. Elsevier, 2006, vol. 3.

[69] G. Hasenjaeger, "Eine bemerkung zu Henkin's beweis für die vollständigkeit des prädikatenkalküls der ersten stufe," *The Journal of Symbolic Logic*, vol. 18, no. 1, pp. 42–48, 1953.

[70] R. W. Quackenbush, "Completeness theorems for universal and implicational logics of algebras via congruences," *Proceedings of the American Mathematical Society*, vol. 103, no. 4, pp. 1015–1021, 1988.

[71] J. Bell and M. Machover, *A course in mathematical logic.* Amsterdam, Netherlands: North Holland, 1977.

[72] C. Berline, "Graph models of λ-calculus at work, and variations," *Mathematical Structures in Computer Science*, vol. 16, no. 2, pp. 185–221, 2006.

[73] G. Manzonetto, "Models and theories of lambda calculus," Ph.D. dissertation, Università Ca' Foscari di Venezia, 2008. [Online]. Available: https://tel.archives-ouvertes.fr/tel-00715207

[74] D. Scott, "Continuous lattices," in *Toposes, Algebraic Geometry and Logic.* Berlin, Heidelberg: Springer, 1972, pp. 97–136.

[75] G. Berry, "Stable models of typed λ-calculi," in *Automata, Languages and Programming.* Berlin, Heidelberg: Springer, 1978, pp. 72–89.

[76] J.-Y. Girard, "The system F of variable types, fifteen years later," *Theoretical Computer Science*, vol. 45, pp. 159–192, 1986. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0304397586900447

[77] A. Bucciarelli and T. Ehrhard, "A theory of sequentiality," *Theoretical Computer Science*, vol. 113, no. 2, pp. 273–291, 1993.

[78] A. Bucciarelli and A. Salibra, "The sensible graph theories of lambda calculus," in *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science (LICS'04).* Turku, Finland: IEEE, July 2004, pp. 276–285.

[79] J.-Y. Girard, "Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur," Ph.D. dissertation, Paris Diderot University, Paris, France, 1972.

[80] J. C. Reynolds, "Towards a theory of type structure," in *Programming Symposium.* Berlin, Heidelberg: Springer, 1974, pp. 408–425.

[81] H. Barendregt, "Lambda calculi with types," in *Handbook of Logic in Computer Science.* UK: Oxford University Press, 1993, vol. 2, background: computational structures, ch. 2, pp. 117–309.

[82] R. Milner, J. Parrow, and D. Walker, “A calculus of mobile processes (part 1),” *Information and Computation*, vol. 100, no. 1, pp. 1–40, 1992. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0890540192900084

[83] A. Popescu and G. Roşu, “Term-generic logic (extended technical report),” Technische Universitat Munchen, University of Illinois at Urbana-Champaign, Tech. Rep., 2013.

[84] L. Cardelli, S. Martini, J. C. Mitchell, and A. Scedrov, “An extension of system F with subtyping,” *Information and Computation*, vol. 109, no. 1, pp. 4–56, 1994.

[85] J.-Y. Girard, “Linear logic,” *Theoretical Computer Science*, vol. 50, no. 1, pp. 1–101, 1987. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0304397587900454

[86] P. Lincoln and J. Mitchell, “Operational aspects of linear lambda calculus,” in *Proceedings of the 7th Annual IEEE Symposium on Logic in Computer Science (LICS’92)*. California, USA: IEEE, June 1992, pp. 235–246.

[87] P. Martin-Löf, *Twenty five years of constructive type theory*, ser. Oxford Logic Guides Book. Oxford, UK: Oxford University Press, 1998, vol. 36, ch. An intuitionistic theory of types, pp. 127–172.

[88] J. Berdine, C. Calcagno, and P. W. O’Hearn, “Symbolic execution with separation logic,” in *Proceedings of the 3rd Asian conference on Programming Languages and Systems (APLAS’05)*, vol. 3780. Tsukuba, Japan: Springer, Nov. 2005, pp. 52–68.

[89] R. Iosif, A. Rogalewicz, and J. Simacek, “The tree width of separation logic with recursive definitions,” in *Proceedings of the 24th International Conference on Automated Deduction (CADE’13)*, vol. 7898. Springer, 2013, pp. 21–38.

[90] W.-N. Chin, C. David, H. H. Nguyen, and S. Qin, “Automated verification of shape, size and bag properties via user-defined predicates in separation logic,” *Journal of Science of Computer Programming*, vol. 77, no. 9, pp. 1006–1036, 2012.

[91] J. Berdine, C. Calcagno, and P. W. O’Hearn, “A decidable fragment of separation logic,” in *Proceedings of the 24th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS’04)*, vol. 3328. Springer, 2004, pp. 97–109.

[92] J. Katelaan, C. Matheja, and F. Zuleger, “Effective entailment checking for separation logic with inductive definitions,” in *Tools and Algorithms for the Construction and Analysis of Systems*, T. Vojnar and L. Zhang, Eds. Cham: Springer International Publishing, 2019, pp. 319–336.

[93] D. Lucanu and G. Roşu, “CIRC: a circular coinductive prover,” in *CALCO*, 2007, pp. 372–378.

[94] G. J. Holzmann, “The model checker SPIN,” *IEEE Trans. Softw. Eng.*, vol. 23, no. 5, pp. 279–295, 1997.

[95] Z. Ésik, "Completeness of Park induction," *Theoretical Computer Science*, vol. 177, no. 1, pp. 217–283, 1997.

[96] M. Sighireanu, J. A. Navarro Pérez, A. Rybalchenko, N. Gorogiannis, R. Iosif, A. Reynolds, C. Serban, J. Katelaan, C. Matheja, T. Noll, F. Zuleger, W.-N. Chin, Q. L. Le, Q.-T. Ta, T.-C. Le, T.-T. Nguyen, S.-C. Khoo, M. Cyprian, A. Rogalewicz, T. Vojnar, C. Enea, O. Lengal, C. Gao, and Z. Wu, "SL-COMP: Competition of solvers for separation logic," in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Beyer, M. Huisman, F. Kordon, and B. Steffen, Eds. Cham: Springer International Publishing, 2019, pp. 116–132.

[97] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08).* Springer, 2008, pp. 337–340.

[98] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11).* Berlin, Heidelberg: Springer, 2011, pp. 171–177.

[99] W. W. Boone, "The word problem," *Proceedings of the National Academy of Sciences*, vol. 44, no. 10, pp. 1061–1065, 1958.

[100] R. Goldblatt, *Logics of Time and Computation*, 2nd ed., ser. CSLI Lecture Notes. Stanford, CA: Center for the Study of Language and Information, 1992, no. 7.

[101] O. Lichtenstein and A. Pnueli, "Propositional temporal logics: Decidability and completeness." *Logic Journal of the IGPL*, vol. 8, no. 1, pp. 55–85, 2000. [Online]. Available: http://dblp.uni-trier.de/db/journals/igpl/igpl8.html#LichtensteinP00

[102] C. Enea, O. Lengál, M. Sighireanu, and T. Vojnar, "Compositional entailment checking for a fragment of separation logic," *Formal Methods in System Design*, vol. 51, no. 3, pp. 575–607, Dec. 2017. [Online]. Available: https://doi.org/10.1007/s10703-017-0289-4

[103] J. Brotherston, N. Gorogiannis, and R. L. Petersen, "A generic cyclic theorem prover," in *Programming Languages and Systems*, R. Jhala and A. Igarashi, Eds. Kyoto, Japan: Springer, 2012, pp. 350–367.

[104] T. F. Şerbănuţă and G. Roşu, "A truly concurrent semantics for the K framework based on graph transformations," in *Proceedings of the 6th International Conference on Graph Transformation (ICGT'12).* Bremen, Germany: Springer, 2012, pp. 294–310.

[105] L. Li and E. Gunter, "IsaK-static A complete static semantics of K," in *Formal Aspects of Component Software.* Springer, 2018, pp. 196–215.

[106] B. Moore, L. Peña, and G. Roşu, "Program verification by coinduction," in *Proceedings of the 27th European Symposium on Programming (ESOP'18).* Springer, 2018, pp. 589–618.

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

[108] J. Goguen and J. Meseguer, "Order-sorted algebra, part I: equational deduction for multiple inheritance, overloading, exceptions and partial operations," *Theoretical Computer Science*, vol. 105, no. 2, pp. 217–273, 1992.

[109] T. Nelson, D. Dougherty, K. Fisler, and S. Krishnamurthi, "On the finite model property in order-sorted logic," Worcester Polytechnic Institute, Brown University, Tech. Rep., 2010.

[110] X. Chen and G. Roşu, "Matching μ-logic," in *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'19).* Vancouver, Canada: IEEE, 2019, pp. 1–13.

[111] C. Barrett, L. De Moura, and P. Fontaine, "Proofs in satisfiability modulo theories," *All about proofs, Proofs for all*, vol. 55, no. 1, pp. 23–44, 2015.

[112] F. Durán and H. Garavel, "The rewrite engines competitions: a RECtrospective," in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Beyer, M. Huisman, F. Kordon, and B. Steffen, Eds. Cham: Springer International Publishing, 2019, pp. 93–100.

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

[114] X. Chen, Z. Lin, M.-T. Trinh, and G. Roşu, "Towards a trustworthy semantics-based language framework via proof generation," in *Proceedings of the 33rd International Conference on Computer-Aided Verification.* Virtual: ACM, July 2021.

[115] G. Roşu, A. Ştefănescu, Ştefan Ciobâcă, and B. M. Moore, "Reachability logic," University of Illinois Urbana-Champaign, Tech. Rep. http://hdl.handle.net/2142/32952, July 2012.

[116] Coq Team, *The Coq proof assistant*, LogiCal Project, 2020. [Online]. Available: http://coq.inria.fr

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

[118] Coq Team, "Coq github repository," https://github.com/coq/coq, 2021.

[119] K Team, "Metamath proof checker in Rust," GitHub page https://github.com/oopsla23-paper-23/k-proof-generation/tree/main/deps/rust-metamath, 2022.

[120] M. Carneiro, "Metamath zero: Designing a theorem prover prover," in *International Conference on Intelligent Computer Mathematics.* Springer, 2020, pp. 71–88.

[121] Tukaani Team, "Xz utils," https://tukaani.org/xz/, 2021.

[122] K Team, "K framework haskell backend," https://github.com/kframework/kore, 2022.

[123] J. Goguen, J. Thatcher, E. Wagner, and J. Wright, "Initial algebra semantics and continuous algebras," *Journal of the ACM*, vol. 24, no. 1, pp. 68–95, 1977.

[124] X. Chen, D. Lucanu, and G. Roşu, "Initial algebra semantics in matching logic," University of Illinois at Urbana-Champaign, Tech. Rep. http://hdl.handle.net/2142/107781, July 2020.

[125] A. Stump, D. Oe, A. Reynolds, L. Hadarean, and C. Tinelli, "Smt proof checking using a logical framework," *Formal Methods in System Design*, vol. 42, no. 1, pp. 91–118, 2013.

[126] A. Ştefănescu, c. Ciobâcă, R. Mereuţă, B. M. Moore, T. F. Şerbănuţă, and G. Roşu, "All-path reachability logic," in *Proceedings of the Joint 25th International Conference on Rewriting Techniques and Applications and 12th International Conference on Typed Lambda Calculi and Applications (RTA-TLCA'14)*, vol. 8560. Springer, 2014, pp. 425–440.

[127] SV-COMP, "Benchmark for sv-comp," https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks, 2021.

[128] S. O'Rear and M. Carneiro, "Metamath verifier in rust," https://github.com/sorear/smetamath-rs, 2019.

[129] G. D. Plotkin, "Lcf considered as a programming language," *Theoretical computer science*, vol. 5, no. 3, pp. 223–255, 1977.

[130] R. Guy, *Unsolved problems in number theory*. Berlin, Heidelberg: Springer Science & Business Media, 2004, vol. 1.

[131] R. Levien and D. A. Wheeler, "Metamath verifier in python," https://github.com/david-a-wheeler/mmverify.py, 2019.

[132] Y. Zhang and B. Xu, "A survey of semantic description frameworks for programming languages," *ACM SIGPLAN Notices*, vol. 39, no. 3, pp. 14–30, 2004. [Online]. Available: http://doi.acm.org.proxy2.library.illinois.edu/10.1145/981009.981013

[133] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual, "CENTAUR: The system," in *Proceedings of the 3rd ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE'88)*. ACM, 1988, pp. 14–24.

[134] P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strniša, "Ott: Effective tool support for the working semanticist," *Journal of Functional Programming*, vol. 20, no. 1, pp. 71–122, 2010.

[135] L. T. van Binsbergen, N. Sculthorpe, and P. D. Mosses, "Tool support for component-based semantics," in *Companion Proceedings of the 15th International Conference on Modularity.* ACM, 2016, pp. 8–11.

[136] M. van den Brand, J. Heering, P. Klint, and P. A. Olivier, "Compiling language definitions: The ASF+SDF compiler," *ACM Transactions on Programming Languages and Systems (TOPLAS'02)*, vol. 24, no. 4, pp. 334–368, 2002.

[137] E. Visser, "Syntax definition for language prototyping," Ph.D. dissertation, University of Amsterdam, 1997.

[138] E. Visser, Z.-e.-A. Benaissa, and A. Tolmach, "Building program optimizers with rewriting strategies," in *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming (ICFP'98).* ACM, 1998, pp. 13–26.

[139] J. Smits and E. Visser, "FlowSpec: Declarative dataflow analysis specification," in *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering (SLE'17).* ACM, 2017, pp. 221–231.

[140] M. Felleisen, R. B. Findler, and M. Flatt, *Semantics engineering with PLT Redex.* Mit Press, 2009.

[141] E. Torlak and R. Bodik, "A lightweight symbolic virtual machine for solver-aided host languages," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*, ser. PLDI '14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2594291.2594340 pp. 530–541.

[142] J. Bornholt and E. Torlak, "Finding code that explodes under symbolic evaluation," *Proceedings of the ACM on Programming Languages*, vol. 2, no. 149, pp. 1–26, 2018.

[143] N. G. de Bruijn, "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem," *Indagationes Mathematicae*, vol. 75, no. 5, pp. 381–392, 1972. [Online]. Available: http://www.sciencedirect.com/science/article/pii/1385725872900340

[144] A. M. Pitts, "Nominal logic, a first order theory of names and binding," *Information and Computation*, vol. 186, no. 2, pp. 165–193, 2003.

[145] F. Pfenning and C. Elliott, "Higher-order abstract syntax," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'88).* New York, NY, USA: ACM, 1988, pp. 199–208.

[146] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy, "Explicit substitutions," *Journal of Functional Programming*, vol. 1, no. 4, pp. 375–416, 1991.

[147] J. W. Klop, "Term rewriting systems," in *Handbook of Logic in Computer Science.* USA: Oxford University Press, Inc., 1993, vol. 2, Background: computational structures, ch. 1, pp. 1–116.

[148] A. M. Pitts, *Nominal sets: names and symmetry in computer science*, ser. Cambridge Tracts in Theoretical Computer Science. New York, NY, USA: Cambridge University Press, 2013.

[149] J. Cheney, "Completeness and Herbrand theorems for nominal logic," *Journal of Symbolic Logic*, vol. 71, no. 1, pp. 299–320, 2006.

[150] J. Cheney, "A simple sequent calculus for nominal logic," *Journal of Logic and Computation*, vol. 26, no. 2, pp. 699–726, 2014.

[151] M. Gabbay and J. Cheney, "A sequent calculus for nominal logic," in *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science (LICS'04)*. Washington, DC, USA: IEEE, 2004, pp. 139–148.

[152] M. Gabbay and A. Pitts, "A new approach to abstract syntax involving binders," in *Proceedings of the 14th Symposium on Logic in Computer Science (LICS'19)*. Trento, Italy: IEEE, July 1999, pp. 214–224.

[153] A. M. Pitts, "Alpha-structural recursion and induction," in *Theorem Proving in Higher Order Logics*, J. Hurd and T. Melham, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 17–34.

[154] C. Urban, "Nominal techniques in Isabelle/HOL," *Journal of Automated Reasoning*, vol. 40, no. 4, pp. 327–356, May 2008. [Online]. Available: https://doi.org/10.1007/s10817-008-9097-2

[155] M. Gabbay and M. Gabbay, "Representation and duality of the untyped λ-calculus in nominal lattice and topological semantics, with a proof of topological completeness," *Annals of Pure and Applied Logic Volume*, vol. 168, no. 3, pp. 501–621, Oct. 2017.

[156] M. Ayala-Rincón, W. de Carvalho-Segundo, M. Fernández, and D. Nantes-Sobrinho, "Nominal C-unification," in *Proceedings of the 27th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'17)*, ser. Lecture Notes in Computer Science, vol. 10855. Namur, Belgium: Springer International Publishing, 2018, pp. 235–251.

[157] M. Ayala-Rincón, M. Fernández, and D. Nantes-Sobrinho, "Nominal narrowing," in *Proceedings of the 1st International Conference on Formal Structures for Computation and Deduction (FSCD'16)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 52. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2016/5983 pp. 1–17.

[158] R. Harper, F. Honsell, and G. Plotkin, "A framework for defining logics," *Journal of the ACM*, vol. 40, no. 1, pp. 143–184, 1993.

[159] R. C. McDowell and D. A. Miller, "Reasoning with higher-order abstract syntax in a logical framework," *ACM Transactions on Computational Logic*, vol. 3, no. 1, pp. 80–136, 2002.

[160] L. C. Paulson, “The foundation of a generic theorem prover,” *Journal of Automated Reasoning*, vol. 5, no. 3, pp. 363–397, 1989. [Online]. Available: https://doi.org/10.1007/BF00248324

[161] A. Felty and A. Momigliano, “Hybrid, a definitional two-level approach to reasoning with higher-order abstract syntax,” *Journal of Automated Reasoning*, vol. 48, no. 1, pp. 43–105, 2012.

[162] A. Gacek, D. Miller, and G. Nadathur, “A two-level logic approach to reasoning about computations,” *Journal of Automated Reasoning*, vol. 49, no. 2, pp. 241–273, 2012. [Online]. Available: https://doi.org/10.1007/s10817-011-9218-1

[163] M. Fiore and O. Mahmoud, “Second-order algebraic theories,” in *Mathematical Foundations of Computer Science 2010*, P. Hliněný and A. Kučera, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 368–380.

[164] F. Pfenning and C. Schürmann, “System description: Twelf—a meta-logical framework for deductive systems,” in *Proceedings of the 16th International Conference on Automated Deduction (CADE 99)*. Trento, Italy: Springer, 1999, pp. 202–206.

[165] J. Cheney, M. Norrish, and R. Vestergaard, “Formalizing adequacy: a case study for higher-order abstract syntax,” *Journal of Automated Reasoning*, vol. 49, no. 2, pp. 209–239, 2012.

[166] D. Kesner, “A theory of explicit substitutions with safe and full composition,” *Logical Methods in Computer Science*, vol. 5, no. 3, pp. 1–29, 2009.

[167] C. J. Bloo, “Preservation of termination for explicit substitution,” Ph.D. dissertation, Technische Universiteit Eindhoven, 1997.

[168] M.-O. Stehr, “CINNI—a generic calculus of explicit substitutions and its application to λ- ς- and ϕ-calculi,” *Electronic Notes in Theoretical Computer Science*, vol. 36, pp. 70–92, 2000.

[169] J. Brotherston and M. Kanovich, “Undecidability of propositional separation logic and its neighbours,” *Journal of the ACM*, vol. 61, no. 2, Apr. 2014. [Online]. Available: https://doi.org/10.1145/2542667

[170] Z. Rakamarić, J. Bingham, and A. J. Hu, “An inference-rule-based decision procedure for verification of heap-manipulating programs with mutable data and cyclic data structures,” in *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI’07)*, vol. 4349. California, USA: Springer, Jan. 2007, pp. 106–121.

[171] Z. Rakamarić, R. Bruttomesso, A. J. Hu, and A. Cimatti, “Verifying heap-manipulating programs in an SMT framework,” in *Proceedings of the 5th International Symposium on Automated Technology for Verification and Analysis (ATVA’07)*, vol. 4762. Tokyo, Japan: Springer, Oct. 2007, pp. 237–252.

[172] S. Lahiri and S. Qadeer, “Back to the future: revisiting precise program verification using SMT solvers,” in *Proceedings of the 35th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL’08)*. ACM, 2008, pp. 171–182.

[173] S. Ranise and C. Zarba, “A theory of singly-linked lists and its extensible decision procedure,” in *Proceedings of the 4th IEEE International Conference on Software Engineering and Formal Methods (SEFM’06)*. IEEE, 2006, pp. 206–215.

[174] A. Bouajjani, C. Drăgoi, C. Enea, and M. Sighireanu, “A logic-based framework for reasoning about composite data structures,” in *Proceedings of the 20th International Conference on Concurrency Theory (CONCUR’09)*, vol. 5710. Springer, 2009, pp. 178–195.

[175] N. Bjørner and J. Hendrix, “Linear functional fixed-points,” in *Proceedings of the 21st International Conference on Computer Aided Verification (CAV’09)*, vol. 5643. Springer, 2009, pp. 124–139.

[176] J. A. N. Pérez and A. Rybalchenko, “Separation logic + superposition calculus = heap theorem prover,” in *Proceedings of the 32nd annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI’11)*. ACM, 2011, pp. 556–566.

[177] R. Piskac, T. Wies, and D. Zufferey, “Automating separation logic using SMT,” in *Proceedings of the 25th International Conference on Computer Aided Verification (CAV’13)*, vol. 8044. Springer, 2013, pp. 773–789.

[178] K. R. M. Leino and M. Moskal, “Co-induction simply,” in *Proceedings of the 19th International Symposium on Formal Methods (FM’14)*, no. 8442. Springer, 2014, pp. 382–398.

[179] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies, “VCC: A practical system for verifying concurrent C,” in *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs’09)*, vol. 5674. Springer, 2009, pp. 23–42.

[180] B. Jacobs, J. Smans, and F. Piessens, “A quick tour of the VeriFast program verifier,” in *Proceedings of the 8th Asian Symposium of Programming Languages and Systems (APLAS’10)*, vol. 6461. Springer, 2010, pp. 304–311.

[181] D.-H. Chu, J. Jaffar, and M.-T. Trinh, “Automatic induction proofs of data-structures in imperative programs,” in *Proceedings of the 36th annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI’15)*. ACM, 2015, pp. 457–466.

[182] H. Unno, S. Torii, and H. Sakamoto, “Automating induction for solving Horn clauses,” in *Proceedings of the 29th International Conference on Computer Aided Verification (CAV’17)*, vol. 10427. Springer, 2017, pp. 571–591.

[183] Q.-T. Ta, T. C. Le, S.-C. Khoo, and W.-N. Chin, "Automated mutual induction proof in separation logic," *Formal Aspects of Computing*, vol. 31, no. 2, pp. 207–230, Apr. 2019.

[184] C. Löding, M. Parthasarathy, and L. Peña, "Foundations for natural proofs and quantifier instantiation," *Proceedings of the ACM on Programming Languages*, vol. 2, no. 10, pp. 1–30, 2017.

[185] J. Brotherston, D. Distefano, and R. L. Petersen, "Automated cyclic entailment proofs in separation logic," in *Proceedings of the 23rd International Conference on Automated Deduction (CAV'11).* Utah, USA: Springer, 2011, pp. 131–146.

[186] D. Baelde, D. Miller, and Z. Snow, "Focused inductive theorem proving," in *Proceedings of the 5th International Joint Conference on Automated Reasoning (IJCAR'10).* Edinburgh, UK: Springer, 2010, pp. 278–292.

[187] X. Leroy, "The CompCert verified compiler, software and commented proof," Available at https://compcert.org/, Mar. 2020.

[188] A. Pnueli, M. Siegel, and E. Singerman, "Translation validation," in *Tools and Algorithms for the Construction and Analysis of Systems*, B. Steffen, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 151–166.

[189] G. Parthasarathy, P. Müller, and A. J. Summers, "Formally validating a practical verification condition generator," in *Computer Aided Verification*, A. Silva and K. R. M. Leino, Eds. Cham: Springer International Publishing, 2021, pp. 704–727.

[190] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens, "Cakeml: a verified implementation of ml," *ACM SIGPLAN Notices*, vol. 49, no. 1, pp. 179–191, 2014.

[191] R. Harper, D. MacQueen, and R. Milner, *Standard ML.* Edinburgh, UK: Department of Computer Science, University of Edinburgh, 1986. [Online]. Available: http://www.lfcs.inf.ed.ac.uk/reports/86/ECS-LFCS-86-2/

[192] K. Slind and M. Norrish, "A brief overview of HOL4," in *International Conference on Theorem Proving in Higher Order Logics*, Springer. Montreal, Canada: Springer-Verlag Berlin Heidelberg, 2008, pp. 28–32.

[193] Q. Garchery, "A framework for proof-carrying logical transformations," *Electronic Proceedings in Theoretical Computer Science*, vol. 336, pp. 5–23, July 2021. [Online]. Available: https://doi.org/10.4204%2Feptcs.336.2

[194] J.-C. Filliâtre and A. Paskevich, "Why3 — where programs meet provers," in *Programming Languages and Systems*, M. Felleisen and P. Gardner, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 125–128.

[195] S. Wils and B. Jacobs, "Certifying c program correctness with respect to compcert with verifast," 2021. [Online]. Available: https://arxiv.org/abs/2110.11034

[196] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens, "Verifast: A powerful, sound, predictable, fast verifier for c and java," in *NASA Formal Methods*, M. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 41–55.

[197] S. Blazy and X. Leroy, "Mechanized semantics for the clight subset of the c language," *Journal of Automated Reasoning*, vol. 43, no. 3, pp. 263–288, 2009.

[198] G. C. Necula and P. Lee, "Proof generation in the touchstone theorem prover," in *Proceedings of the 17th International Conference on Automated Deduction*, Springer. Pittsburgh, Pennsylvania, USA: Springer-VerlagBerlin, Heidelberg, 2000, pp. 25–44.