Towards a Unified Proof Framework for Automated Fixpoint Reasoning using Matching Logic

XIAOHONG CHEN, University of Illinois at Urbana-Champaign, USA
MINH-THAI TRINH, Advanced Digital Sciences Center, Illinois at Singapore, Singapore
NISHANT RODRIGUES, University of Illinois at Urbana-Champaign, USA
LUCAS PEÑA, University of Illinois at Urbana-Champaign, USA
GRIGORE ROŞU, University of Illinois at Urbana-Champaign, USA

Automation of fixpoint reasoning has been extensively studied for various mathematical structures, logical formalisms, and computational domains, resulting in specialized fixpoint provers for heaps, for streams, for term algebras, for temporal properties, for program correctness, and for many other formal systems and inductive and coinductive properties. However, in spite of great theoretical and practical interest, there is no unified framework for automated fixpoint reasoning. Although several attempts have been made, there is no evidence that such a unified framework is possible, or practical. In this paper, we propose a candidate based on *matching logic*, a formalism recently shown to *theoretically* unify the above mentioned formal systems. Unfortunately, the (KNASTER-TARSKI) proof rule of matching logic, which enables inductive reasoning, is not syntax-driven. Worse, it can be applied at any step during a proof, making automation seem hopeless. Inspired by recent advances in automation of inductive proofs in separation logic, we propose an alternative proof system for matching logic, which is amenable for automation. We then discuss our implementation of it, which although not superior to specialized state-of-the-art automated provers for specific domains, we believe brings some evidence and hope that a unified framework for automated reasoning is not out of reach.

CCS Concepts: • **Theory of computation** → **Automated reasoning**; **Proof theory**.

Additional Key Words and Phrases: matching logic, automated reasoning, fixpoints, induction

ACM Reference Format:
Xiaohong Chen, Minh-Thai Trinh, Nishant Rodrigues, Lucas Peña, and Grigore Roşu. 2020. Towards a Unified Proof Framework for Automated Fixpoint Reasoning using Matching Logic. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 161 (November 2020), 29 pages. https://doi.org/10.1145/3428229

1 INTRODUCTION

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 [Berdine et al. 2004, 2005; Brotherston et al. 2014; Chin et al. 2012; Iosif et al. 2013; Katelaan et al. 2019], for streams [Lucanu and Roşu 2007], for term algebras [Kovács et al.

Authors' addresses: Xiaohong Chen, Department of Computer Science, University of Illinois at Urbana-Champaign, 201 N. Goodwin Ave., Urbana, Illinois, 61801, USA, xc3@illinois.edu; Minh-Thai Trinh, Advanced Digital Sciences Center, Illinois at Singapore, 1 Create Way, Create Tower, Singapore, 138602, Singapore, minhthai.t@adsc-create.edu.sg, trinhmt@illinois.edu; Nishant Rodrigues, Department of Computer Science, University of Illinois at Urbana-Champaign, 201 N. Goodwin Ave., Urbana, Illinois, 61801, USA, nishant2@illinois.edu; Lucas Peña, Department of Computer Science, University of Illinois at Urbana-Champaign, 201 N. Goodwin Ave., Urbana, Illinois, 61801, USA, lpena7@illinois.edu; Grigore Roşu, Department of Computer Science, University of Illinois at Urbana-Champaign, 201 N. Goodwin Ave., Urbana, Illinois, 61801, USA, grosu@illinois.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2020 Copyright held by the owner/author(s).
2475-1421/2020/11-ART161
https://doi.org/10.1145/3428229

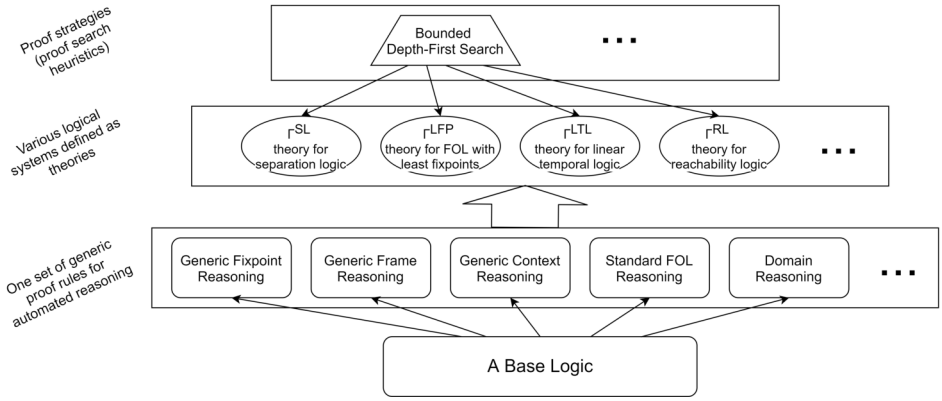

Fig. 1. The Vision of An Ideal Unified Proof Framework for Automated Reasoning

2017], for temporal properties [Holzmann 1997], for program reachability correctness [Roşu et al. 2013], and for many other systems and inductive/coinductive properties. However, in spite of great theoretical and practical interest, there is no unified framework for automated fixpoint reasoning.

Fig. 1 illustrates our vision of an *ideal* unified automated proof framework for fixpoint reasoning. A unified proof framework must be based on a powerful base logic, in which all logical systems and programming languages can be defined as logical theories. 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 the base logic, independently of the specific logical theory. 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 language L in the base 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.

In this paper we present a prototype implementation of a unified proof framework for automating fixpoint reasoning. As base logic we choose *matching logic*, which was recently proposed as a foundation for a variety of logical systems, including FOL with least fixed-points, modal and temporal logics, separation logics, etc. [Chen and Roşu 2019; Roşu 2017]. Besides expressiveness, an advantage that matching logic offers is its compact syntax and convenient notation through its *patterns* (Section 3), which allow us to encode formulae in other logical systems almost verbatim. For example, the matching logic encoding of modal logic defines a symbol $\Diamond\_$ which allows us to encode modal logic formula $\Diamond p$ as matching logic pattern $\Diamond p$, with essentially zero representational distance; this is in sharp contrast to how modal logic is encoded in FOL [Blackburn et al. 2001], e.g., where a binary predicate needs to be added and quantifiers in the resulting FOL formula.

To evaluate our unified proof framework and prototype implementation, we consider four representative logical systems for fixpoint reasoning: (1) first-order logic extended with least fixpoints [Gurevich and Shelah 1985], abbreviated LFP; (2) separation logic extended with recursive definitions [Chin et al. 2012; Reynolds 2002], abbreviated SL; (3) linear temporal logic [Pnueli 1977], abbreviated LTL; and (4) reachability logic [Roşu et al. 2013], abbreviated RL. 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 [Holzmann 1997]. RL is a language-parametric generalization of Hoare logic [Ştefănescu et al. 2016, Section 4], 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 separation logic (SL) in this paper, however, because it gathered much attention in recent years that resulted in several automated SL provers and its own international competition SL-COMP'19 [Sighireanu et al. 2019].

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'19 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" $\varphi_1 \; U \; \varphi_2$ (whose semantics are defined as fixpoints) 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.

An Overview of Our Unified Proof Framework. Our unified proof framework consists of three main reasoning modules: fixpoint, frame, and context (also illustrated in Fig. 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 matching logic theories. Therefore, they accomplish fixpoint reasoning, frame reasoning, and context reasoning for *all* logical systems defined as theories in matching logic.

The main challenge we faced while developing our unified proof framework was that the existing proof system of matching logic [Chen and Roşu 2019] is too fine-grained to be amenable for automation. For example, its (MODUS PONENS) proof rule "$\vdash \varphi \to \psi$ and $\vdash \varphi$ implies $\vdash \psi$" requires the prover to guess a premise φ, which does not bode well with automation. Most importantly, its (KNASTER-TARSKI) proof rule (also called Park induction [Ésik 1997]) 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 LHS of the proof goal is a standalone least fixpoint. It cannot be directly applied to proof goals in LFP or SL, such as $ll(x,y) * list(y) \to list(x)$ (see Section 4.1.2), where the LHS $C[ll(x,y)]$ contains the fixpoint $ll(x,y)$ within a context $C[h] \equiv h * list(y)$. An indirect application is possible *in theory*, but it involves sophisticated, ad-hoc reasoning to eliminate the context C from the LHS, which cannot be efficiently automated.

Our fixpoint module addresses the above challenge by proposing a context-driven fixpoint proof rule, (KT), shown in Fig. 2b, which is a sequential composition of several proof rules that first (WRAP) context C within the RHS ψ, written $C \multimap \psi$, and eliminate it from the LHS, then apply inductive reasoning, and finally (UNWRAP) context C and restore it on the LHS. The pattern $C \multimap \psi$, called *contextual implication*, is expressible in matching 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 proof goal $\vdash C[\varphi] \to C[\psi]$ 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 4.

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'19 benchmark shows

that the naive bounded-DFS strategy can prove 90% of the properties without frame reasoning, and 95% with frame reasoning (Section 6). This was surprising, because it would place our generic proof framework in the third place in the SL-COMP'19 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.

Organization of the Rest of the Paper. We discuss related work on automated fixpoint reasoning in Section 2. We introduce matching logic and its encodings of SL, LTL, and RL in Section 3. We introduce our unified proof framework in Section 4 and discuss our implementation in Section 5. We show the experimental results in Section 6 and conclude the paper in Section 7.

All proof details can be found in the companion technical report [Chen et al. 2020b].

2 RELATED WORK

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 [Reynolds 2002], which resulted in several automatic separation logic provers; see [Sighireanu et al. 2019] for those that participated in the recent SL-COMP'19 competition. Since separation logic is undecidable [Brotherston and Kanovich 2014], many provers implement only decision procedures to decidable fragments [Berdine et al. 2004; Brotherston et al. 2014; Enea et al. 2017; Katelaan et al. 2019] or incomplete algorithms [Berdine et al. 2005; Chin et al. 2012; Iosif et al. 2013]. There is also work on decision procedures for other heap logics [Bjørner and Hendrix 2009; Bouajjani et al. 2009; Lahiri and Qadeer 2008; Rakamarić et al. 2007a,b; Ranise and Zarba 2006], 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 [Barrett et al. 2011; De Moura and Bjørner 2008] into separation logic provers [Pérez and Rybalchenko 2011; Piskac et al. 2013].

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 [Iosif et al. 2013]. 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 $list(x)$, 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 4.1, 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 [Leino and Moskal 2014], VCC [Cohen et al. 2009] and Verifast [Jacobs et al. 2010]. 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., [Cohen et al. 2009, Section 6] mentions several tools that are based on Coq [The Coq development team 2004] or Isabelle [The Isabelle development team 2018]). 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 [Ştefănescu et al. 2016], 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 4.3.5, 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 [Brotherston et al. 2011, 2012; Chu et al. 2015; Löding et al. 2017; Ta et al. 2019; Unno et al. 2017]. Tac [Baelde et al. 2010] 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. [Brotherston et al. 2012] 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 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.

3 MATCHING LOGIC PRELIMINARIES

We recall the preliminaries of matching logic. Many-sorted matching logic was firstly proposed in [Roşu 2017] and the support for fixpoints was added later in [Chen and Roşu 2019]. The proof framework in this paper aims at the many-sorted matching logic as proposed in [Chen and Roşu 2019]. Recently, [Chen et al. 2020a; Chen and Roşu 2020] proposed a new functional variant of matching logic, whose reasoning will be considered in the future work. Here, we simply call many-sorted matching logic as *matching logic*, abbreviated ML. We shall keep the presentation informal and intuitive unless necessary. Technical details can be found in the mentioned citations.

3.1 An Informal Overview of Matching Logic

Matching logic (ML) can be summarized by the following equation:

Matching Logic =	A unified syntax of *patterns*	+	A unified semantics based on *pattern matching*	+	One fixed Hilbert *proof system*

ML formulas are called *patterns*, which are built from variables, symbols, logical connectives such as $\wedge$ and $\vee$, FOL-style quantification $\exists x$ and $\forall x$, and two constructs μ and ν for the least/greatest fixpoints (formalized in Section 3.2). In particular, there is no distinction made between terms and formulas like in FOL, giving ML the flexibility to subsume, unchanged, the various syntaxes of formulas, assertions, expressions, etc. in FOL, separation logic (SL), modal logics, temporal logics, reachability logic, and more. This syntactic generality may surprise at first sight, but it is a critical feature that makes ML an expressive logic to uniformly specify and reason about properties expressed in various logical systems in their original notation, without awkward encodings.

ML has a *pattern matching* semantics. Intuitively, a pattern φ is interpreted as the set $\|\varphi\|$ of elements that *match* φ. For example, $cons(x, cons(y, nil))$ is a pattern matched by (and only by) the list consisting of x and y, so $\|cons(x, cons(y, nil))\|$ is always a singleton set. Pattern $cons(x, cons(y, nil)) \wedge x \neq y$ additionally states that x is different from y, and its interpretation is either a singleton set (if $x \neq y$) or the empty set (if $x = y$). Note how the "term" $cons(x, cons(y, nil))$ is constrained by a logical condition $x \neq y$; a (likely awkward) encoding would be required to express the same in FOL.

Separation logic (SL) is a good example to illustrate the pattern matching semantics of ML. SL defines heap formulas that evaluate to true or false on a heap h. Heaps can be composed using a binary $\_ * \_$ construct over formulae (not terms). This semantics is properly captured by ML by letting $\|\varphi\|$ be the set of heaps on which φ evaluates to true in SL. [Roşu 2017, Section 9] shows that this treatment of SL heap formulas as ML patterns *verbatim et litteratim* indeed yields the correct, intended semantics. We show some examples below (more formal details are in Section 3.2):

- emp is a pattern matched by the empty heap;
- $x \mapsto y$ is a pattern matched by the singleton heap from x to y if x is nonzero and by no heaps if otherwise; in other words, its semantics depends on the valuations of x (and y);
- $x \mapsto z * y \mapsto z$ is a pattern matched by a 2-entry heap if $x \neq y$, and by no heaps if otherwise.

Note that there is a big, fundamental difference between SL and ML in how they support the semantics (models) of heaps and their formal reasoning. In SL, the notion of heaps is built into the logic, into its syntax, semantics, and proof system. This makes SL "a specialized logic for heaps", and different models or notions of heaps require inventions of different SL variants. In contrast, ML is one fixed logic, with fixed syntax, semantics, and proof system. The heap constructors emp, $\_\mapsto\_$, $\_*\_$ are treated the same as, e.g., the list constructors $cons$ and nil, which are all symbols in the unified pattern syntax and have no pre-defined, built-in semantics. Their semantics is instead *axiomatized* by the *logical theories*, whose formal reasoning is provided by the one fixed proof system of ML. In that sense, ML is a more general-purpose logic than SL, making it motivating to study its formal reasoning and develop its automated provers because they can be applied to not only one specialized theory, but all theories. That SL assertions can be regarded as ML patterns is a plus, suggesting that existing proof techniques used by SL provers can be adopted and generalized to reason about ML patterns and therefore, go beyond the scope of heap reasoning. We shall revisit and elaborate on this point in Section 4 when we present our proof framework.

3.2 Matching Logic Syntax and Semantics: Formal Definitions

Here, we formalize the intuition in Section 3.1 and define the syntax and semantics of ML. This subsection is compact and may look dense to the reader who is not already familiar with ML, but it makes this paper self-contained. We refer the reader to [Chen and Roşu 2019, 2020; Roşu 2017] for details; the rest of the paper is accessible with the level of detail about ML discussed below.

Definition 3.1. Given a *signature* (S, Σ) that consists of a sort set S and an $(S^* \times S)$-indexed family $\Sigma = \{\Sigma_{s_1 \ldots s_n, s}\}_{s_1, \ldots, s_n, s \in S}$ of many-sorted symbols, ML syntax defines *patterns* as follows[1]:

$$\varphi ::= \underbrace{x \mid \sigma(\varphi_1, \ldots, \varphi_n)}_{\text{structures}} \mid \underbrace{\varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi \mid \varphi_1 = \varphi_2 \mid \varphi_1 \subseteq \varphi_2}_{\text{logical constraints}} \mid \underbrace{\exists x.\, \varphi \mid \forall x.\, \varphi}_{\text{quantification}} \mid \underbrace{X \mid \mu X.\, \varphi \mid \nu X.\, \varphi}_{\text{fixpoints}}$$

We divide the above syntax into four groups. *Structures* are built from element variables (denoted $x, y, \ldots$) and symbols, like FOL terms, but with a more flexible, pattern matching semantics (see

[1]We list more constructs than necessary, because we use them in the rest of the paper and we are not trying to be minimal here. As seen in [Chen and Roşu 2019], only two logical constructs ($\wedge$ and $\neg$), one quantifier, and one fixpoint are necessary.

Definition 3.2(2)). For readability, we may use the mixfix syntax for symbols; e.g., we write $\varphi_1 * \varphi_2$ for the binary symbol $\_*\_$ of separating conjunction. *Logical constraints* are built from equations, inclusions, and standard propositional connectives. $\varphi_1 \subseteq \varphi_2$ states that all elements matching φ_1 match φ_2. A common form of patterns is $\varphi_{structure} \wedge \varphi_{constraint}$, which states both the structure and the constraint that the structure should satisfy. *Quantification* can be applied to structures or constraints. For constraints, it is the same as FOL quantification. For structures, it creates *data abstraction*. For example, $\exists z.\ (x \mapsto z * y \mapsto z)$ is matched by *any* heap where x and y point to the same (unspecified) value z; that is, the data z is abstracted away from the pattern. Finally, *fixpoints* are built from set variables (denoted $X, Y, \ldots$, different from element variables) and constructs μ and ν. Intuitively, $\mu X.\ \varphi$ (resp. $\nu X.\ \varphi$) denotes the least (resp. greatest) set X that satisfies the equation $X = \varphi$, where X may recursively occur in φ. To guarantee that the fixpoints exist, we enforce the usual syntactic requirement that X has no negative occurrences in φ; see, e.g., [Kozen 1982].

The ML syntax above includes more constructs than necessary, because we use them in the rest of the paper and we are not trying to be minimal here. As seen in [Chen and Roşu 2020], only two logical constructs are necessary ($\bot$ and $\rightarrow$), one quantifier, and one fixpoint.

ML *semantics* defines models, symbol interpretations, and pattern interpretations. A *model* M is a nonempty underlying carrier set equipped with an interpretation $\sigma_M : M_{s_1} \times \cdots \times M_{s_n} \rightarrow \mathcal{P}(M_s)$ of any symbol $\sigma \in \Sigma_{s_1 \ldots s_n, s}$, where $M_{s_1}, \ldots, M_{s_n}, M_s$ are the carrier sets of sorts $s_1, \ldots, s_n, s$, respectively, and $\mathcal{P}(M_s)$ denotes the powerset of M_s. Formally,

Definition 3.2. Given a signature (S, Σ), an ML *model* consists of a nonempty carrier set M and an interpretation $\sigma_M : M_{s_1} \times \cdots \times M_{s_n} \rightarrow \mathcal{P}(M_s)$ for each $\sigma \in \Sigma_{s_1 \ldots s_n, s}$, where $\mathcal{P}(M_s)$ is the powerset of M_s. Given a *valuation* ρ that maps element variables to elements in M and set variables to subsets of M, we define *pattern interpretation* $\|\varphi\|_{M,\rho}$ inductively as follows:

(1) $\|x\|_{M,\rho} = \{\rho(x)\}$
(2) $\|\sigma(\varphi_1, \ldots, \varphi_n)\|_{M,\rho} = \bigcup_{a_i \in \|\varphi_i\|_{M,\rho}} \sigma_M(a_1, \ldots, a_n)$
(3) $\|\varphi_1 \wedge \varphi_2\|_{M,\rho} = \|\varphi_1\|_{M,\rho} \cap \|\varphi_2\|_{M,\rho}$
(4) $\|\varphi_1 \vee \varphi_2\|_{M,\rho} = \|\varphi_1\|_{M,\rho} \cup \|\varphi_2\|_{M,\rho}$
(5) $\|\neg\varphi\|_{M,\rho} = M \setminus \|\varphi\|_{M,\rho}$
(6) $\|\varphi_1 = \varphi_2\|_{M,\rho} = M$ if $\|\varphi_1\|_{M,\rho} = \|\varphi_2\|_{M,\rho}$
(7) $\|\varphi_1 = \varphi_2\|_{M,\rho} = \emptyset$ if $\|\varphi_1\|_{M,\rho} \neq \|\varphi_2\|_{M,\rho}$
(8) $\|\varphi_1 \subseteq \varphi_2\|_{M,\rho} = M$ if $\|\varphi_1\|_{M,\rho} \subseteq \|\varphi_2\|_{M,\rho}$
(9) $\|\varphi_1 \subseteq \varphi_2\|_{M,\rho} = \emptyset$ if $\|\varphi_1\|_{M,\rho} \not\subseteq \|\varphi_2\|_{M,\rho}$
(10) $\|\exists x.\ \varphi\|_{M,\rho} = \bigcup_{a \in M} \|\varphi\|_{M,\rho[a/x]}$
(11) $\|\forall x.\ \varphi\|_{M,\rho} = \bigcap_{a \in M} \|\varphi\|_{M,\rho[a/x]}$
(12) $\|X\|_{M,\rho} = \rho(X)$
(13) $\|\mu X.\ \varphi\|_{M,\rho} = l.f.p.(\lambda A \mapsto \|\varphi\|_{M,\rho[A/X]})$
(14) $\|\nu X.\ \varphi\|_{M,\rho} = g.f.p.(\lambda A \mapsto \|\varphi\|_{M,\rho[A/X]})$

Definition 3.2 is not unexpected. Note that in (6)-(9), equations/inclusions hold if they evaluate to M and do not hold if they evaluate to $\emptyset$; that is, we use M to denote "true" and $\emptyset$ to denote "false". We shall see it again when we define axioms and validity in ML. The rest of the cases are normal: (1) and (12) interpret element/set variables according to ρ. (2) interprets structure patterns according to symbol interpretations. (3)-(5) interpret propositional connectives as the corresponding set-theoretic operations. (10)-(11) interpret quantification by ranging over x, where $\rho[a/x]$ denotes the updated valuation. (13)-(14) interpret fixpoint patterns as the fixpoints in the model.

ML uses a *logical theory*, or simply a *theory* Γ, which is a set of patterns called *axioms*, to restrict the models and symbol interpretations, by enforcing all axioms in Γ to evaluate to "true" (i.e., total set M). For example, the following pattern/axiom enforces that the binary symbol/operation $*$ is commutative: $h_1 * h_2 = h_2 * h_1$. We say that a pattern/property ψ holds in a model M, or M validates ψ, written $M \vDash \psi$, iff ψ evaluates to the total set M under all valuations. Given a set Γ of axioms, we write $M \vDash \Gamma$ iff $M \vDash \psi$ for all $\psi \in \Gamma$, and $\Gamma \vDash \psi$ iff $M \vDash \psi$ for all $M \vDash \Gamma$. In the following, we show an example theory Γ^{SL} that captures separation logic and its semantics and formal reasoning.

Example: Separation Logic in Matching Logic. As an example, we show that separation logic (SL) models can be formulated as ML models, and the heap assertions/formulas of separation logic can be represented by ML patterns that yield the same semantics (see also [Roşu 2017, Section 9]).

Let us define the signature $(S^{\mathsf{SL}}, \Sigma^{\mathsf{SL}})$, where $S^{\mathsf{SL}} = \{Nat, Map\}$ has two sorts for natural numbers and (finite) maps, respectively, and Σ^{SL} includes the basic arithmetic operations and three map constructors: $emp \in \Sigma^{\mathsf{SL}}_{\epsilon,Map}$ for the empty map, $\mapsto \in \Sigma^{\mathsf{SL}}_{Nat\ Nat,Map}$ for building the singleton maps, and $* \in \Sigma^{\mathsf{SL}}_{Map\ Map,Map}$ for merging two disjoint maps (i.e., separating conjunction). Now, we consider a particular ML model M whose carrier sets M_{Nat} and M_{Map} are the set $\mathbb{N}$ of natural numbers and the set $[\mathbb{N}_{\geq 1} \rightharpoonup_{\mathsf{fin}} \mathbb{N}]$ of finite maps, i.e., finite-domain partial functions from nonzero numbers (locations) to numbers. We define the symbol interpretation in M in the following, expected way: emp has interpretation $emp_M = \{\cdot_{Map}\}$, where $\cdot_{Map} : \mathbb{N}_{\geq 1} \rightharpoonup_{\mathsf{fin}} \mathbb{N}$ is the partial function that is undefined everywhere, denoting the empty map; $\mapsto$ has interpretation $\mapsto_M (x, y) \subseteq M_{Map}$ for any $x, y \in \mathbb{N}$, often written in the mixfix form $x \mapsto_M y$, which is defined to be the partial function mapping x to y and undefined everywhere else if $x \neq 0$, or $\cdot_{Map}$ if $x = 0$; $*$ has interpretation $*_M(h_1, h_2) \subseteq M_{Map}$ for any $h_1, h_2 \in M_{Map}$, or written $h_1 *_M h_2$, which is defined to be the merge (disjoint union) of h_1 and h_2 if their domains are disjoint, or $\cdot_{Map}$ if otherwise. We call such model M of signature $(S^{\mathsf{SL}}, \Sigma^{\mathsf{SL}})$ the *standard map model*. In [Roşu 2017, Section 9], the authors showed that heap assertions are ML patterns of signature $(S^{\mathsf{SL}}, \Sigma^{\mathsf{SL}})$ and their separation logic semantics coincide with their interpretations in the standard map model M; that is, given an SL formula φ, a finite map (heap) h, and a valuation ρ mapping variables in φ to natural numbers, h satisfies φ under ρ iff in the standard map model M, $h \in \|\varphi\|_{M,\rho}$.

Properties about the standard model M can be specified using ML axioms. For example, the axioms $0 \mapsto x = \bot$ and $x \mapsto y * x \mapsto z = \bot$ specify nonzero locations and disjoint map union, which we will see in Section 4.3.1. In [Chen and Roşu 2019, Section 5], the authors showed how to define *recursive symbols* using ML patterns as axioms. For example, we can define ML symbols $ll, lr \in \Sigma^{\mathsf{SL}}_{Nat\ Nat,Map}$ and $list \in \Sigma^{\mathsf{SL}}_{Nat,Map}$ with the following axioms:

$$ll(x, y) =_{\mathrm{lfp}} (x = y \wedge emp) \vee (x \neq y \wedge \exists t.\, x \mapsto t * ll(t, y))$$
$$lr(x, y) =_{\mathrm{lfp}} (x = y \wedge emp) \vee (x \neq y \wedge \exists t.\, lr(x, t) * t \mapsto y)$$
$$list(x) =_{\mathrm{lfp}} (x = nil \wedge emp) \vee (x \neq nil \wedge \exists y.\, x \mapsto y \wedge list(y))$$

where the notation $=_{\mathrm{lfp}}$ means, intuitively, that the interpretation ll_M (similarly for lr_M and $list_M$) is the least one that satisfies the equation among all interpretations $[\mathbb{N} \times \mathbb{N} \to \mathcal{P}(M_{Map})]$. Recall that the μ construct in ML can only build *sets*, so we need to transform ("de-sugar") symbol interpretations to sets, using the equivalence $[\mathbb{N} \times \mathbb{N} \to \mathcal{P}(M_{Map})] \simeq \mathcal{P}(\mathbb{N} \times \mathbb{N} \times M_{Map})$, and thus reduce the task of defining recursive symbols to defining least fixpoints of $\mathcal{P}(M \times M \times M)$ using μ. Full technical details about the automatic de-sugaring of $=_{\mathrm{lfp}}$ into μ are in [Chen and Roşu 2019], but are not necessary to understand the results presented in this paper.

3.3 The Hilbert Proof System and Its Limitations on Automated Reasoning

A Hilbert-style ML proof system that defines the provability relation $\Gamma \vdash_H \varphi$ of a given theory Γ and pattern/pattern φ was proposed in [Chen and Roşu 2019, Fig. 1]. As a high-level overview, we list some important meta-theorems about the formal reasoning carried out by the proof system.

Proposition 3.3. *For any Γ and φ, the following propositions hold:*

(1) $\Gamma \vdash_H \varphi$, *if φ is a propositional tautology over patterns;*
(2) $\Gamma \vdash_H \varphi_1$ *and* $\Gamma \vdash_H \varphi_1 \to \varphi_2$ *imply* $\Gamma \vdash_H \varphi_2$, *known as the* (MODUS PONENS) *rule;*
(3) $\Gamma \vdash_H \varphi[y/x] \to \exists x.\, \varphi$;
(4) $\Gamma \vdash_H \varphi_1 \to \varphi_2$ *and* $y \notin \mathrm{FV}(\varphi_2)$ *imply* $\Gamma \vdash_H (\exists y.\, \varphi_1) \to \varphi_2$;

(5) $\Gamma \vdash_H \varphi = \varphi$;
(6) $\Gamma \vdash_H \varphi_1 = \varphi_2$ *and* $\Gamma \vdash_H \varphi_2 = \varphi_3$ *imply* $\Gamma \vdash_H \varphi_1 = \varphi_3$;
(7) $\Gamma \vdash_H \varphi_1 = \varphi_2$ *implies* $\Gamma \vdash_H \varphi_2 = \varphi_1$;
(8) $\Gamma \vdash_H \varphi_1 = \varphi_2$ *implies* $\Gamma \vdash_H \psi[\varphi_1/x] = \psi[\varphi_2/x]$, *known as the Leibniz property of equality;*
(9) $\Gamma \vdash_H \varphi[(\mu X.\, \varphi)/X] = \mu X.\, \varphi$;
(10) $\Gamma \vdash_H \varphi[\psi/X]$ *implies* $\Gamma \vdash_H \mu X.\, \varphi \to \psi$, *known as the* (KNASTER-TARSKI) *rule or Park induction.*

In other words, the Hilbert proof system supports standard propositional/FOL reasoning, standard equational reasoning, and standard fixpoint reasoning. We review the *soundness theorem* below:

THEOREM 3.4 (SOUNDNESS THEOREM [CHEN AND ROŞU 2019]). $\Gamma \vdash_H \varphi$ *implies* $\Gamma \vDash \varphi$.

We point out that the Hilbert system, although important and interesting from a foundational aspect, is *not* practical for automated reasoning because it gives too many degrees of freedom in proof search. For example, its (MODUS PONENS) rule does not bode well with automation because the premise φ needs to be guessed. More importantly, the fixpoint reasoning rule (KNASTER-TARSKI) requires the LHS of the proof goal be a least fixpoint, and thus cannot be applied directly to recursive predicates and structures or when the least fixpoint occurs within a context. All the above make proof automation based on the Hilbert system unfeasible. In Section 4, we discuss our main technical contribution, which is a proof framework that is most suitable for proof automation.

3.4 Important Logics Defined as Matching Logic Theories

The main motivation of this paper is to propose an automated proof framework for ML. By defining various logical systems as theories in ML, we capture the various forms of formal reasoning by one proof framework in a unified way. In this subsection, we consider three typical logical systems that involve fixpoint reasoning and discuss how they can be defined as ML theories. They are separation logic extended with user-defined recursive predicates [Iosif et al. 2013] (abbreviated SL), linear temporal logic [Pnueli 1977] (abbreviated LTL), and reachability logic [Roşu et al. 2013] (abbreviated RL) for semantics-based formal verification (of which Hoare-style verification [Hoare 1969] is an instance). We use these three logical systems to demonstrate the generality of our prover.

3.4.1 Separation Logic (SL). We already discussed SL in Section 3.2. Here, we only mention that *recursive symbols* in SL can take the following general form:

$$p(\tilde{x}) =_{\text{lfp}} \exists \tilde{x}_1.\, \varphi_1(\tilde{x}, \tilde{x}_1) \vee \cdots \vee \exists \tilde{x}_m.\, \varphi_m(\tilde{x}, \tilde{x}_m)$$

where $\tilde{x}_i$ is a vector of variables. Each $\exists \tilde{x}_i.\, \varphi_i(\tilde{x}, \tilde{x}_i)$ is called a *case*. If p does not occur in φ_i then the case is a *base case*. Otherwise, it is an *inductive case*. For example, in the following definition $(x = y \wedge emp)$ is the base case and $(x \neq y \wedge \exists t.\, x \mapsto t * ll(t, y))$ is the inductive case:

$$ll(x, y) =_{\text{lfp}} (x = y \wedge emp) \vee (x \neq y \wedge \exists t.\, x \mapsto t * ll(t, y))$$

The ML theory Γ^{SL} axiomatizes the algebraic properties about heap constructs and defines recursive heap predicates as recursive symbols. When our generic prover is used to prove SL properties, it will be instantiated by Γ^{SL} that includes the following axioms:

<u>ML theory for Separation Logic</u>
Sorts S^{SL}: Nat, Map
Symbols Σ^{SL}: $emp \in \Sigma^{\mathsf{SL}}_{\epsilon, Map}$ $\quad \_\mapsto\_ \in \Sigma^{\mathsf{SL}}_{Nat\ Nat, Map}$ $\quad \_*\_ \in \Sigma^{\mathsf{SL}}_{Map\ Map, Map}$
recursive heap predicate $p \in \Sigma^{\mathsf{SL}}_{Nat \ldots Nat, Map}$
Axioms Γ^{SL}:
$h_1 * (h_2 * h_3) = (h_1 * h_2) * h_3 \quad h_1 * h_2 = h_2 * h_1 \quad emp * h = h \quad x \mapsto y \to x \neq nil$
$x_1 \mapsto y * x_2 \mapsto z \to x_1 \neq x_2 \quad p(\tilde{x}) =_{\text{lfp}} \exists \tilde{x}_1.\, \varphi_1(\tilde{x}, \tilde{x}_1) \vee \cdots \vee \exists \tilde{x}_m.\, \varphi_m(\tilde{x}, \tilde{x}_m)$

Another core SL construct, which we have not emphasized much but which is important and challenging in formal reasoning, is the *separating implication* $\varphi_1 \mathrel{-\!\!*} \varphi_2$, also known as the "magic wand". Its semantics is the inverse of separating conjunction $*$, in the sense that $(\varphi_1 * \varphi_2) \to \psi$ iff $\varphi_1 \to (\varphi_2 \mathrel{-\!\!*} \psi)$. Many proof systems and provers for SL rely on the magic wand. In ML, we can define a more general concept, called *contextual implication* and discussed in Section 4.1.2, and show that magic wand is a special instance. Although in ML contextual implication can be axiomatized, i.e., it is *not* an extension of the logic, we found it to be a very practical and intuitive instrument. We will propose the context reasoning module, which has a set of (automatic) proof rules to reason about contextual implication. As it turns out, contextual implication can be used to guide our automatic prover through the goal pattern to where inductive reasoning is actually needed. Its usage is far beyond the scope of SL and has occurred in almost all our examples.

3.4.2 Linear Temporal Logic (LTL). LTL is an important temporal logic for specifying and reasoning about infinite execution traces. Its syntax extends propositional logic with a set of *temporal operators* including "next" $\circ\varphi$, "always" $\Box\varphi$, "eventually" $\Diamond\varphi$, "until" $\varphi_1 \; U \; \varphi_2$, etc. Intuitively, $\circ\varphi$ holds iff φ holds on the next state; $\Box\varphi$ holds iff φ always holds; $\Diamond\varphi$ holds iff φ eventually holds; and $\varphi_1 \; U \; \varphi_2$ holds iff φ_2 eventually holds and before that φ_1 holds.

As shown in [Chen and Roşu 2019, Section 7], we need only one sort $State$ for states and one (unary) ML symbol $\bullet\_ \in \Sigma^{\mathsf{LTL}}_{State,State}$, called *one-path next*, to define LTL as a theory in ML. We explain the idea intuitively. Consider any transition system and a state s. Let $\bullet s$ be the pattern matched by those states whose next states include s. In other words, $\bullet s$ is matched by all predecessor states of s, because, by definition, they can transit to s in one step, so "one-path next s" holds.

The one-path next $\bullet$ encodes the entire transition relation, so we can use it to build patterns that express various temporal properties. For example $\bullet\varphi$ is matched by the states which have at least a next state that matches φ, i.e., "on one path 'next φ' holds". Its dual is the "all-path next", defined by $\circ\varphi \equiv \neg\bullet\neg\varphi$, which is matched by the states whose nest states all match φ. In LTL models, the next-state relation is a function, so $\bullet$ and $\circ$ coincide. This is easily captured by an ML pattern/axiom $\bullet\varphi = \circ\varphi$. The other LTL constructs can be defined from $\bullet$ (and $\circ$) as fixpoints in the usual way, and thus the following definitions are standard:

<u>ML theory for Linear Temporal Logic</u>
Sort S^{LTL}: $State$
Symbol Σ^{LTL}: one-path next $\bullet \in \Sigma^{\mathsf{LTL}}_{State,State}$
Axioms Γ^{LTL}:
$\circ\varphi = \neg\bullet\neg\varphi \quad \Diamond\varphi =_{\mathrm{lfp}} \varphi \vee \bullet\Diamond\varphi \quad \Box\varphi =_{\mathrm{gfp}} \varphi \wedge \circ\Box\varphi \quad \varphi_1 \; U \; \varphi_2 =_{\mathrm{lfp}} \varphi_2 \vee \varphi_1 \wedge \bullet(\varphi_1 \; U \; \varphi_2)$
$\bullet\varphi = \circ\varphi \quad \circ(\varphi_1 \wedge \varphi_2) = \circ\varphi_1 \wedge \circ\varphi_2$ referred as $(\circ\wedge)$

3.4.3 Reachability Logic (RL). RL is an approach to program verification based on operational semantics. Its main difference from the traditional Hoare-style verification is that RL uses one fixed proof system, the reachability proof system, to achieve sound and relatively complete deduction for all languages. Thus, RL can be pragmatically seen as a generic Hoare logic, where the target programming language is plug-and-played. RL has been used to define the complete formal semantics of several large languages such as C [Hathhorn et al. 2015], Java [Bogdănaş and Roşu 2015], and JavaScript [Park et al. 2015], as well as of emerging blockchain languages such as the EVM [Hildenbrandt et al. 2018], yielding program verifiers for all these languages automatically [Ştefănescu et al. 2016]. The RL formulae, called *(reachability) rules* and written $\varphi_1 \Rightarrow \varphi_2$ where φ_1, φ_2 are patterns, mean that for all program configurations γ matching φ_1, either γ has an infinite execution trace or it reaches a configuration matching φ_2 (partial correctness).

Fixing a RL semantics, the coresponding ML theory Γ^{RL}, like Γ^{LTL}, also uses the one-path next $\bullet$ symbol to capture the transition relation on configurations and axiomatizes the following reachability constructs (we only show a part; see [Chen and Roşu 2019] for full details):

ML theory for Reachability Logic
Sorts S^{RL}: *Cfg* for configurations and sorts for programs, data, and environments (omitted)
Symbols Σ^{RL}: one-path next $\bullet \in \Sigma^{\mathsf{RL}}_{Cfg,Cfg}$ and constructors of data and environments
Axioms Γ^{RL}
$\mathsf{WF} = \mu X.\, \circ X$ (well-founded states)
$\varphi_1 \Rightarrow \varphi_2 = \varphi_1 \to (\mathsf{WF} \to \Diamond\varphi_2)$ referred as (REACH)
$\circ\varphi_1 \wedge \bullet\varphi_2 \to \bullet(\varphi_1 \wedge \varphi_2)$ referred as ($\circ\bullet$)

4 AUTOMATED PROOF FRAMEWORK FOR MATCHING LOGIC

In this section, we propose our automated proof framework for ML and a set of automatic proof rules (Fig. 2) that accomplish the five reasoning modules as illustrated in Fig. 1. As we discussed in Section 3.3, the existing Hilbert proof system of ML is not suitable for automated reasoning. Our main contribution in this paper is the proposal of a new set of higher-level proof rules (Fig. 2) that aim at proof automation. The generic ML prover simply runs a simple bounded depth-first search algorithm over the proposed set of proof rules. We will give an overview of the three key reasoning modules offered by the automated proof rules in Section 4.1 and then explain all proof rules in detail in Section 4.2. In Section 4.3, we use several examples to show how our proof framework can be applied to various logical theories.

4.1 Reasoning Modules

Our proof framework consists of three main reasoning modules: fixpoint reasoning module, context reasoning module, and frame reasoning module (also illustrated in Fig. 1). In this subsection, we give an intuitive introduction to these three modules. We write $\Gamma \vdash \varphi$ to mean that φ can be proved by the proof framework within theory Γ, and $\vdash \varphi$ when Γ is understood or irrelevant.

4.1.1 Fixpoint Reasoning Module and the Core Fixpoint Rule (LFP). 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 ($\tilde{x}, \tilde{x}_1, \ldots, \tilde{x}_m$ denote variable vectors):

$$p(\tilde{x}) =_{\mathrm{lfp}} \exists \tilde{x}_1.\, \varphi_1(\tilde{x}, \tilde{x}_1) \vee \cdots \vee \exists \tilde{x}_m.\, \varphi_m(\tilde{x}, \tilde{x}_m)$$

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 Fig. 2b):

$$(\text{LFP}) \quad \frac{\exists \tilde{x}_1.\, \varphi_1[\psi/p] \to \psi \quad \cdots \quad \exists \tilde{x}_m.\, \varphi_m[\psi/p] \to \psi}{p(\tilde{x}) \to \psi} \tag{1}$$

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 RHS. 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 LHS).

4.1.2 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 separation logic (SL) example (where ll and $list$ are defined at the end of Section 3.2):

$$\vdash ll(x, y) * list(y) \rightarrow list(x) \tag{2}$$

Clearly, (LFP) cannot be applied directly to (2), because the LHS is not a recursive symbol, but a larger pattern $ll(x, y) * list(y)$ in which the recursive pattern $ll(x, y)$ occurs. In other words, $ll(x, y)$ occurs *within a context* in the LHS. Let $C[h] \equiv h * list(y)$ be the *context pattern* where h is a distinguished hole variable. We rewrite proof goal (2) to the following form using context C:

$$\vdash C[ll(x, y)] \rightarrow list(x) \tag{3}$$

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., $C_{id}[h] \equiv h$, 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.

Contextual Implications. To solve the above challenge, we propose an important concept called *contextual implication*. In the following, we first give a formal definition of context patterns and contextual implications and then revisit the above SL example.

Definition 4.1. A *context pattern* C or simply *context* is a pattern with a distinguished variable denoted h, called the hole variable. We write $C[\varphi]$ as the substitution $C[\varphi/h]$. Given an ML theory Γ, we say that C is a *structure context* w.r.t. Γ if $C \equiv t \wedge \psi$ where t is a structure pattern and ψ is a predicate, and h only occurs in t within nested symbols (and not other logical constructs). All contexts considered in this paper are structure contexts.

In other words, a context C is a structure context if the hole variable h occurs only within nested structures. For example, $h * list(y) \wedge y > 1$ is a structure context (w.r.t. h) because separating conjunction $*$ is an ML symbol. 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 $C[h]$ be a structure context, and ψ be some property. We define *contextual implication* w.r.t. C and ψ as the pattern whose matching elements, if plugged into C, satisfy ψ. Formally

Definition 4.2. We define *contextual implication* $C \multimap \psi \equiv \exists h.\, h \wedge (C[h] \subseteq \psi)$.

Recall Definition 3.2, where the semantics of $\exists$ 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:

$$\vdash C[\varphi] \rightarrow \psi \quad \underset{\text{(UNWRAP) context } C}{\overset{\text{(WRAP) context } C}{\rightleftharpoons}} \quad \vdash \varphi \rightarrow (C \multimap \psi)$$

Note that contextual implication $C \multimap \psi$ is a normal ML pattern defined using the ML syntax in Section 3. It is *not an extension of ML*, but simply a convenient use of the existing expressiveness of ML patterns to simplify (and automate) formal reasoning by "pulling the target out of its context".

Now, we revisit the SL example at the beginning of this subsection and look at proof goal (3). By wrapping the structure context $C[h] = h * list(y)$, we transform it to the following equivalent goal, to which (LFP) can be applied:

$$\vdash ll(x, y) \to (C \multimap list(x)) \quad \text{where } C[h] = h * list(y)$$

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 RHS becomes $C \multimap \psi$, which by (LFP) will be moved back to LHS and replace the recursive occurrences of the recursive pattern (see Eq. (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 LHS using pattern matching. This is explained in detail in Section 4.2.

The second remark is that our contextual implication generalizes separating implication $\varphi \mathrel{-\!\!*} \psi$ (the "magic wand") in SL. Indeed, let context $C_\varphi[h] = h * \varphi$, then we have $\varphi \mathrel{-\!\!*} \psi = C_\varphi \multimap \psi$. In other words, SL magic wand is a special instance of ML contextual implication, where the underlying theory is Γ^{SL} (see Section 3.4.1) and context $C[h]$ has the specific form $h * \varphi$ where h occurs immediately below the top-level $*$ operator, and the SL proof rule (ADJ) [Reynolds 2002, 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 $C[h]$, e.g., to entire program configurations (see Section 4.3.5) not only heaps. Also, the (WRAP) and (UNWRAP) proof rules generalize the (ADJ) proof rule in SL.

4.1.3 Frame Reasoning within Any Contexts. Another advantage of having an explicit notion of context as shown above, is that *frame reasoning* can be generalized to all contexts C (the mild technical conditions mentioned in Section 4.1.2 are sufficient for its soundness). In the following, we compare the frame reasoning in separation logic for heap contexts (left, also called (MONOTONE) in [Reynolds 2002]) and the general frame reasoning in matching logic for any contexts C (right):

$$\text{(FRAME) in SL} \quad \frac{\varphi \to \psi}{\varphi * \varphi_{rest} \to \psi * \varphi_{rest}} \qquad\qquad \text{(FRAME) in ML} \quad \frac{\varphi \to \psi}{C[\varphi] \to C[\psi]}$$

Clearly, (FRAME) in SL is a special instance of (FRAME) in ML, where context $C[h] \equiv h * \varphi_{rest}$ is a heap context. ML (FRAME) 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:

$$\text{(FRAME ON PROGRAMS)} \quad \frac{\varphi \ \{\mathsf{code}\} \ \psi}{\varphi * \varphi_{rest} \ \{\mathsf{code}\} \ \psi * \varphi_{rest}} \quad \text{if no variable free in } \varphi_{rest} \text{ is modified by } \mathsf{code}$$

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 $\varphi * \varphi_{rest}$ $\{\mathsf{code}\}$ $\psi * \varphi_{rest}$ 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 ML is universally sound for all logical theories and thus programming

$$(\textsc{elim-}\exists)\ \frac{\varphi \to \psi}{(\exists x.\, \varphi) \to \psi}\ \text{if } x \notin \mathrm{FV}(\psi)$$

$$(\textsc{smt})\ \frac{\texttt{True}}{\varphi \to \psi}\ \text{if } \vDash_{\mathsf{SMT}} \varphi \to \psi$$

$$(\textsc{match-ctx})\ \frac{C_{rest}[\varphi'\theta] \to \psi}{C_o[\forall \tilde{y}.\, (C' \multimap \varphi')] \to \psi}\ \text{where } (C_{rest}, \theta) = \mathrm{cm}(C_o, C', \tilde{y})$$

$$(\textsc{pm})\ \frac{\varphi \to \psi\theta}{\varphi \to \exists \tilde{y}.\, \psi}\ \text{where } \theta \in \mathtt{pm}(\varphi, \psi, \tilde{y}) \text{ matches } \varphi \text{ with } \psi$$

$$(\textsc{frame})\ \frac{\varphi \to \psi}{C[\varphi] \to C[\psi]}$$

$$(\textsc{unfold-r})\ \frac{\varphi \to C[\varphi_i]}{\varphi \to C[p(\tilde{x})]}$$

$$(\textsc{kt})\ \frac{\text{Composition of Rules in Fig. 2b}}{\varphi \to \psi}$$

(a) Proof Rules for ML Fixpoint Reasoning

$$(\textsc{wrap})\ \frac{p(\tilde{x}) \to (C \multimap \psi)}{C[p(\tilde{x})] \to \psi}$$

$$(\textsc{intro-}\forall)\ \frac{p(\tilde{x}) \to \forall \tilde{y}.\ (C \multimap \psi)}{p(\tilde{x}) \to (C \multimap \psi)}\ \text{where } \tilde{y} = \mathrm{FV}(\psi) \setminus \tilde{x}$$

$$(\textsc{lfp})\ \frac{\cdots\ \varphi_i[\forall \tilde{y}.\, (C \multimap \psi)/p] \to \forall \tilde{y}.\, (C \multimap \psi)}{p(\tilde{x}) \to \forall \tilde{y}.\, (C \multimap \psi)}$$

$$(\textsc{elim-}\forall)\ \frac{\varphi \to (C \multimap \psi)}{\varphi \to \forall y.\, (C \multimap \psi)}\ \text{if } y \notin \mathrm{FV}(\varphi)$$

$$(\textsc{unwrap})\ \frac{C[\varphi] \to \psi}{\varphi \to (C \multimap \psi)}$$

(b) Breakdown of Rule (KT) in Fig. 2a

Fig. 2. Automatic Proof Framework for ML Fixpoint Reasoning (where $p(\tilde{x}) =_{\mathsf{lfp}} \bigvee_i \varphi_i$)

languages whose semantics are defined as ML 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.

4.2 Framework Description

Here we discuss our automated proof rules in Fig. 2a, where the (KT) rule is a composition of (WRAP), (LFP), and (UNWRAP) as shown in Fig. 2b. The generic proof framework is *parametric* in an ML theory Γ, and it proves implications, i.e., $\Gamma \vdash \varphi \to \psi$. Formally:

Definition 4.3. If $\varphi \to \psi$ can be proved using the proof rules in Fig. 2 within the underlying theory Γ, we write $\Gamma \vdash \varphi \to \psi$, abbreviated $\vdash \varphi \to \psi$ if Γ is understood.

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, containing no logical constraints, quantification, or fixpoints (see ML syntax in Section 3). A *conjunctive (resp. disjunctive) pattern* is a pattern of the form $\varphi_1 \wedge \cdots \wedge \varphi_n$ (resp. $\varphi_1 \vee \cdots \vee \varphi_n$), where $\varphi_1, \ldots, \varphi_n$ are structure patterns. In Fig. 2, we assume p is a recursive symbol defined by $p(\tilde{x}) =_{\mathsf{lfp}} \bigvee_i \varphi_i$ where each φ_i denotes one definition case.

➢ (ELIM-$\exists$) is a standard FOL rule that simplifies the LHS by removing existential variables. Note that the side condition $x \notin \mathrm{FV}(\psi)$ 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-$\exists$) exhaustively, we can obtain a LHS that is quantifier-free at the top.

➢ (SMT) does domain reasoning using SMT solvers such as Z3 [De Moura and Bjørner 2008] and CVC4 [Barrett et al. 2011], 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 $\vDash_{\texttt{SMT}} \varphi \to \psi$ 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 RHS. The algorithm pm will be discussed in Section 5. 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 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 4.3.1 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 [Boone 1958], 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 RHS within any context C (satisfying mild conditions for contextual implication in Section 4.1.2) following its definition $p(\tilde{x}) =_{\text{lfp}} \bigvee_i \varphi_i$. 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 [Tarski 1955], is a sequential composition of five proof rules shown in Fig. 2b: (WRAP), (INTRO-$\forall$), (LFP), (ELIM-$\forall$), and (UNWRAP). We explained the core proof rule (LFP) in Section 4.1.1. We explained in Section 4.1.2 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 RHS and thus makes the subsequent proofs easier, because the (strengthened) RHS will be moved to the LHS by (LFP). Then after (LFP), we apply (ELIM-$\forall$) to restore the RHS 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 RHS are contextual implications, because those contextual implications are moved to the LHS by (LFP) and then block the proofs, because (so far) we have not defined any proof rules that can handle contextual implications on the LHS. 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 LHS 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 4.1.1) 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 LHS 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 5. Here, we give the key intuition behind it.

When can a contextual implication $C' \multimap \psi'$ be eliminated? Recall Definition 4.2, 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:

$$\vdash C'[C' \multimap \psi'] \rightarrow \psi' \tag{4}$$

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, property (4) can be used to handle contextual implication on the LHS. 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 property (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 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.

Soundness. We conclude by the soundness of the proof rules in Fig. 2, which is proved by the following theorem stating that these rules are provable using the Hilbert proof system in Section 3.3.

THEOREM 4.4. *If φ is provable from Γ using the proof rules in Fig. 2, written $\Gamma \vdash \varphi$, then φ is provable from Γ using the Hilbert proof system (Section 3.3) plus the proof rule* (SMT).

PROOF. The complete proof can be found in the companion technical report [Chen et al. 2020b]. In short, the proof rules (ELIM-∃), (PM), (INTRO-∀), (ELIM-∀) can be proved by standard FOL reasoning, which are supported by the Hilbert system (see Proposition 3.3). Rules (LFP) and (UNFOLD-R) can be proved by standard fixpoint reasoning, also supported by the Hilbert system. Rules (FRAME), (MATCH-CTX), (WRAP), and (UNWRAP) rely on the properties of structure contexts. □

Combining Theorem 4.4 with Theorem 3.4, we conclude that our proof framework is sound, assuming that the SMT solvers used in the proof rule (SMT) are sound.

THEOREM 4.5. *If φ is provable from Γ using the proof rules in Fig. 2, then $\Gamma \vDash \varphi$, assuming the soundness of the SMT solvers used in the proof rule* (SMT).

4.3 Examples

We have so far explained our proof rules. In this subsection, we show how these rules are put into practice by using them to prove several example proof goals collected from the various logical systems mentioned in Section 3. Our objective is to help the reader understand better our proof framework and some subtle technical details, to show that the proof rules in Fig. 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.

$$
\begin{array}{rl}
\textsc{smt} & \dfrac{\texttt{True}}{lr(x,w) * w \mapsto y \wedge z \neq y \wedge x \neq y \rightarrow x \neq y \wedge lr(x,w) * w \mapsto y} \\
\textsc{pm} & \dfrac{lr(x,w) * w \mapsto y \wedge z \neq y \wedge x \neq y \rightarrow x \neq y \wedge \exists t.\, lr(x,t) * t \mapsto y}{lr(x,w) * w \mapsto y \wedge z \neq y \wedge x \neq y \rightarrow lr(x,y)} \\
\textsc{unfold-r} & \dfrac{lr(x,w) * w \mapsto y \wedge z \neq y \wedge x \neq y \rightarrow lr(x,y)}{x \mapsto z * (\forall x.\, (C' \multimap lr(x,w))) * w \mapsto y \wedge z \neq y \wedge x \neq y \rightarrow lr(x,y) \quad (\dagger)} \\
\textsc{match-ctx} & \dfrac{x \mapsto z * (\forall x.\, (C' \multimap lr(x,w))) * w \mapsto y \wedge z \neq y \wedge x \neq y \rightarrow lr(x,y) \quad (\dagger)}{\exists w.\, x \mapsto z * (\forall x.\, (C' \multimap lr(x,w))) * w \mapsto y \wedge z \neq y \wedge x \neq y \rightarrow lr(x,y)} \\
\textsc{elim-}\exists & \dfrac{\exists w.\, x \mapsto z * (\forall x.\, (C' \multimap lr(x,w))) * w \mapsto y \wedge z \neq y \wedge x \neq y \rightarrow lr(x,y)}{\exists w.\, (\forall x.\, (C' \multimap lr(x,w))) * w \mapsto y \wedge z \neq y \rightarrow (C \multimap lr(x,y))} \\
\textsc{unwrap} & \dfrac{\exists w.\, (\forall x.\, (C' \multimap lr(x,w))) * w \mapsto y \wedge z \neq y \rightarrow (C \multimap lr(x,y))}{\exists w.\, (\forall x.\, (C' \multimap lr(x,w))) * w \mapsto y \wedge z \neq y \rightarrow \forall x.\, (C \multimap lr(x,y))} \\
\textsc{elim-}\forall & \dfrac{\exists w.\, (\forall x.\, (C' \multimap lr(x,w))) * w \mapsto y \wedge z \neq y \rightarrow \forall x.\, (C \multimap lr(x,y)) \qquad \cdots}{lr(z,y) \rightarrow \forall x.\, (C \multimap lr(x,y))} \\
\textsc{lfp} & \dfrac{lr(z,y) \rightarrow \forall x.\, (C \multimap lr(x,y))}{lr(z,y) \rightarrow (C \multimap lr(x,y))} \\
\textsc{intro-}\forall & \dfrac{lr(z,y) \rightarrow (C \multimap lr(x,y))}{x \mapsto z * lr(z,y) \wedge x \neq y \rightarrow lr(x,y)} \\
\textsc{wrap} & \dfrac{x \mapsto z * lr(z,y) \wedge x \neq y \rightarrow lr(x,y)}{\exists z.\, x \mapsto z * lr(z,y) \wedge x \neq y \rightarrow lr(x,y)} \\
\textsc{elim-}\exists & \dfrac{\exists z.\, x \mapsto z * lr(z,y) \wedge x \neq y \rightarrow lr(x,y) \qquad \cdots}{ll(x,y) \rightarrow lr(x,y)} \\
\textsc{lfp} &
\end{array}
$$

Fig. 3. Proof tree of $\vdash ll(x,y) \rightarrow lr(x,y)$, where $C[h] \equiv x \mapsto z * h \wedge x \neq y$ and $C'[h] \equiv x \mapsto z * h \wedge x \neq w$.

4.3.1 A Basic Example from Separation Logic. We first prove $\vdash ll(x,y) \rightarrow lr(x,y)$, (Example 2 in [Chu et al. 2015]), where

$$
\begin{aligned}
ll(x,y) &=_{\text{lfp}} (x = y \wedge emp) \vee (x \neq y \wedge \exists t.\, x \mapsto t * ll(t,y)) \\
lr(x,y) &=_{\text{lfp}} (x = y \wedge emp) \vee (x \neq y \wedge \exists t.\, lr(x,t) * t \mapsto y)
\end{aligned}
$$

The proof tree is shown in Fig. 3. Since the LHS $ll(x,y)$ 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 $ll(x,y)$:

$$\vdash (x = y \wedge emp) \rightarrow lr(x,y)$$

The other goal corresponds to the inductive case and is shown in the second last line in Fig. 3. For clarity, we breakdown the steps in calculating the substitution $[lr(x,y)/ll]$ required by (LFP) below:

$\vdash ll(x,y) \rightarrow lr(x,y)$	proof goal, before (LFP) is applied
$\vdash (\exists z.\, x \mapsto z * ll(z,y) \wedge x \neq y) \rightarrow lr(x,y)$	phantom step 1: unfolding to inductive case
$\vdash (\exists z.\, x \mapsto z * lr(z,y) \wedge x \neq y) \rightarrow lr(x,y)$	phantom step 2: substituting lr for ll

Now, the base case goal can be proved by applying (UNFOLD-R) to unfold the RHS $lr(x,y)$ to its base case and then calling SMT solvers. The inductive case (after eliminating $\exists z$ from LHS), $\vdash x \mapsto z * lr(z,y) \wedge x \neq y \rightarrow lr(x,y)$, contains a recursive pattern $lr(z,y)$ within a context $C[h] = x \mapsto z * h \wedge x \neq y$. Therefore, we (WRAP) the context and yield contextual implication $C \multimap lr(x,y)$ on the RHS, 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:

$\vdash lr(z,y) \rightarrow (C \multimap lr(x,y))$	proof goal, before (LFP) is applied
$\vdash (\exists w.\, lr(z,w) * w \mapsto y \wedge z \neq y) \rightarrow (C \multimap lr(x,y))$	phantom step 1: unfolding
$\vdash (\exists w.\, (\forall x.\, (C \multimap lr(x,y)))[w/y] * w \mapsto y \wedge z \neq y) \rightarrow (C \multimap lr(x,y))$	phantom step 2: substituting

where $(\forall x.\, (C \multimap lr(x,y)))[w/y] = \forall x.\, (C' \multimap lr(x,w))$ and $C'[h] = x \mapsto z * h \wedge x \neq w$.

Now the proof proceeds by (UNWRAP)-ping the context C on the RHS and moving it back to the LHS, and eliminating the quantifier $\exists w$ by (ELIM-$\exists$). Then the proof goal becomes the following

$$
\begin{array}{rl}
\textsc{smt} & \dfrac{\texttt{True}}{lr(x,w,s_3) * \phi \rightarrow lr(x,w,s_3) * w \mapsto y \wedge s{=}s_3{\cup}\{w\} \wedge x \neq y} \\
\textsc{pm} & \dfrac{lr(x,w,s_3) * \phi \rightarrow lr(x,w,s_3) * w \mapsto y \wedge s{=}s_3{\cup}\{w\} \wedge x \neq y}{lr(x,w,s_3) * \phi \rightarrow \exists t \exists s_4.\, lr(x,t,s_4) * t \mapsto y \wedge s{=}s_4{\cup}\{t\} \wedge x \neq y} \\
\textsc{unfold-r} & \dfrac{lr(x,w,s_3) * \phi \rightarrow \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 \rightarrow lr(x,y,s)} \\
\textsc{match-ctx} & \dfrac{lr(x,w,s_3) * \phi \rightarrow lr(x,y,s)}{x \mapsto z * (\forall x \forall s.\, (C' \multimap lr(x,w,s))) * \phi \rightarrow lr(x,y,s) \qquad (\ddagger)} \\
\textsc{elim-}\exists & \dfrac{x \mapsto z * (\forall x \forall s.\, (C' \multimap lr(x,w,s))) * \phi \rightarrow 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 \rightarrow lr(x,y,s)} \\
\textsc{unwrap} & \dfrac{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 \rightarrow 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 \rightarrow (C \multimap lr(x,y,s))} \\
\textsc{elim-}\forall & \dfrac{\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 \rightarrow (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 \rightarrow \forall x \forall s.\, (C \multimap lr(x,y,s))} \\
\textsc{lfp} & \dfrac{\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 \rightarrow \forall x \forall s.\, (C \multimap lr(x,y,s)) \qquad \cdots}{lr(z,y,s_1) \rightarrow \forall x \forall s.\, (C \multimap lr(x,y,s))} \\
\textsc{intro-}\forall & \dfrac{lr(z,y,s_1) \rightarrow \forall x \forall s.\, (C \multimap lr(x,y,s))}{lr(z,y,s_1) \rightarrow (C \multimap lr(x,y,s))} \\
\textsc{wrap} & \dfrac{lr(z,y,s_1) \rightarrow (C \multimap lr(x,y,s))}{x \mapsto z * lr(z,y,s_1) \wedge s{=}s_1{\cup}\{x\} \wedge x \neq y \rightarrow lr(x,y,s)} \\
\textsc{elim-}\exists & \dfrac{x \mapsto z * lr(z,y,s_1) \wedge s{=}s_1{\cup}\{x\} \wedge x \neq y \rightarrow 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 \rightarrow lr(x,y,s)} \\
\textsc{lfp} & \dfrac{\exists z \exists s_1.\, x \mapsto z * lr(z,y,s_1) \wedge s{=}s_1{\cup}\{x\} \wedge x \neq y \rightarrow lr(x,y,s) \qquad \cdots}{ll(x,y,s) \rightarrow lr(x,y,s)}
\end{array}
$$

$$
\begin{array}{ll}
\text{where} & C[h] \equiv x \mapsto z * h \wedge s = s_1 \cup \{x\} \wedge x \neq y \\
& C'[h] \equiv C[h][w/y, s_2/s_1] = x \mapsto z * h \wedge s = s_2 \cup \{x\} \wedge x \neq w \\
& \phi \equiv w \mapsto y \wedge s_1 = s_2 \cup \{w\} \wedge z \neq y \wedge s = s_1 \cup \{x\} \wedge x \neq y
\end{array}
$$

Fig. 4. Proof tree of $\vdash ll(x,y,s) \rightarrow lr(x,y,s)$

(formula (†) in line 5, Fig. 3):

$$x \mapsto z * (\forall x.\, (C' \multimap lr(x,w))) * w \mapsto y \wedge z \neq y \wedge x \neq y \rightarrow lr(x,y)$$

At this point, the quantified contextual implication on the LHS is instantiated and matched by (MATCH-CTX), which calls the context matching algorithm cm, introduced in Section 5. Intuitively, the algorithm uses heuristics to produce an instantiation for $\forall x$ (in 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 SL axiom $x_1 \mapsto y * x_2 \mapsto z \rightarrow x_1 \neq x_2$ given in Section 4.3.1. Therefore, (MATCH-CTX) is applied successfully, and the rest, unmatched context of C_o is left in the goal (line 4 of Fig. 3) and proved in the subsequent proofs.

4.3.2 A Slightly More Complex Example from Separation Logic. The previous simple example does not illustrate the usage of (INTRO-∀), because (MATCH-CTX) applied to goal (†) in Fig. 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:

$$ll(x,y,s) =_{\text{lfp}} (x = y \wedge emp \wedge s = \emptyset) \vee \exists x_1 \exists s_1.\, x{\mapsto}x_1 * ll(x_1,y,s_1) \wedge s{=}s_1{\cup}\{x\} \wedge x{\neq}y$$
$$lr(x,y,s) =_{\text{lfp}} (x = y \wedge emp \wedge s = \emptyset) \vee \exists y_1 \exists s_1.\, lr(x,y_1,s_1) * y_1{\mapsto}y \wedge s{=}s_1{\cup}\{y_1\} \wedge x{\neq}y$$

Its proof tree in Fig. 4 is similar to the one in Fig. 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 Fig. 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, instead of having ‡, we will have

$$x \mapsto z * (C' \multimap lr(x,w,s)) * w \mapsto y \wedge s_1 = s_2 \cup \{w\} \wedge z \neq y \wedge s = s_1 \cup \{x\} \wedge x \neq y \rightarrow lr(x,y,s)$$

$$
\textsc{wrap}\;\dfrac{\textsc{intro-}\forall\;\dfrac{\textsc{lfp}\;\dfrac{\textsc{elim-}\forall\;\dfrac{\textsc{unwrap}\;\dfrac{\textsc{elim-}\exists\;\dfrac{\textsc{match-ctx}\;\dfrac{\textsc{unfold-r}\;\dfrac{\textsc{pm}\;\dfrac{\textsc{smt}\;\dfrac{\text{True}}{x \mapsto t_1 * t_1 \mapsto t_2 * \mathit{llE}(t_2,z) \to x \mapsto t_1 * t_1 \mapsto t_2 * \mathit{llE}(t_2,z)}}{x \mapsto t_1 * t_1 \mapsto t_2 * \mathit{llE}(t_2,z) \to \exists u_1 \exists u_2 .\, x \mapsto u_1 * u_1 \mapsto u_2 * \mathit{llE}(u_2,z)}}{x \mapsto t_1 * t_1 \mapsto t_2 * \mathit{llE}(t_2,z) \to \mathit{llE}(x,z)}}{x \mapsto t_1 * t_1 \mapsto t_2 * (\forall z.\,(C \multimap \mathit{llE}(t_2,z))) * \mathit{llO}(y,z) \to \mathit{llE}(x,z)}}{\exists t_1 \exists t_2 .\, x \mapsto t_1 * t_1 \mapsto t_2 * (\forall z.\,(C \multimap \mathit{llE}(t_2,z))) * \mathit{llO}(y,z) \to \mathit{llE}(x,z)}}{\exists t_1 \exists t_2 .\, x \mapsto t_1 * t_1 \mapsto t_2 * \forall z.\,(C \multimap \mathit{llE}(t_2,z)) \to (C \multimap \mathit{llE}(x,z))}}{\exists t_1 \exists t_2 .\, x \mapsto t_1 * t_1 \mapsto t_2 * \forall z.\,(C \multimap \mathit{llE}(t_2,z)) \to (C \multimap \mathit{llE}(x,z))} \qquad \cdots}{\mathit{llO}(x,y) \to \forall z.\,(C \multimap \mathit{llE}(x,z))}}{\mathit{llO}(x,y) \to (C \multimap \mathit{llE}(x,z))}}{\mathit{llO}(x,y) * \mathit{llO}(y,z) \to \mathit{llE}(x,z)}
$$

Fig. 5. Proof tree of $\vdash \mathit{llO}(x,y) * \mathit{llO}(y,z) \to \mathit{llE}(x,z)$, where $C[h] \equiv h * \mathit{llO}(y,z)$

where $C'[h] = x \mapsto z * h \wedge s = s_2 \cup \{x\} \wedge x \neq w$. 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.

4.3.3 A Mutual Recursion Example from Separation Logic. Mutually recursive definitions are in general defined as:

$$
\begin{cases}
p_1(\tilde{y}_1) =_{\text{lfp}} \exists \tilde{x}_{11} .\, \varphi_{11}(\tilde{y}_1, \tilde{x}_{11}) \vee \cdots \vee \exists \tilde{x}_{1m_1} .\, \varphi_{1m_1}(\tilde{y}_1, \tilde{x}_{1m_1}) \\
\qquad \cdots \\
p_k(\tilde{y}_k) =_{\text{lfp}} \exists \tilde{x}_{k1} .\, \varphi_{k1}(\tilde{y}_k, \tilde{x}_{k1}) \vee \cdots \vee \exists \tilde{x}_{km_k} .\, \varphi_{km_k}(\tilde{y}_k, \tilde{x}_{km_k})
\end{cases}
$$

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 `qf_shid_entl/10.tst.smt2` from the SL-COMP'19 competition [Sighireanu et al. 2019] as an example. Consider the following definition of list segments of odd and even length:

$$
\begin{cases}
\mathit{llO}(x,y) =_{\text{lfp}} x \mapsto y \vee \exists t .\, x \mapsto t * \mathit{llE}(t,y) \\
\mathit{llE}(x,y) =_{\text{lfp}} \exists t .\, x \mapsto t * \mathit{llO}(t,y)
\end{cases}
$$

and the proof goal $\vdash \mathit{llO}(x,y) * \mathit{llO}(y,z) \to \mathit{llE}(x,z)$.

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.

$$
\mathit{llO}(x,y) =_{\text{lfp}} x \mapsto y \vee \exists t_1 \exists t_2 .\, x \mapsto t_1 * t_1 \mapsto t_2 * \mathit{llO}(t_2,y)
$$
$$
\mathit{llE}(x,y) =_{\text{lfp}} \exists t_1 \exists t_2 .\, x \mapsto t_1 * t_1 \mapsto t_2 * \mathit{llE}(t_2,y)
$$

Then, the proof can be carried out in the normal way. We show the proof tree in Fig. 5.

4.3.4 A Linear Temporal Logic (LTL) Example. The purpose of this and the next examples is to demonstrate the generality of our proof method. In this subsection, we show an example of proving the induction proof rule of the sound and complete proof system of LTL [Goldblatt 1992; Lichtenstein and Pnueli 2000], which uses the ML axiomatization for LTL given in Section 4.3.4 and the greatest fixpoint reasoning rules that are dual to those in Fig. 2. The key dual rule (GFP) will be shown explicitly. In the next subsection, we show an example of proving the (partial) correctness of a simple `sum` program that computes the total from 1 to a symbolic input n, which shows that formal verification is also a form of fixpoint reasoning and can be achieved by our proof method.

$$\textsc{gfp}\ \dfrac{\textsc{pm}\ \dfrac{\circ\wedge\ \dfrac{\textsc{unfold-L}\ \dfrac{\textsc{smt}\ \dfrac{\texttt{True}}{p \wedge (p \to \circ p) \wedge \circ\Box(p \to \circ p) \to \circ p \wedge \circ\Box(p \to \circ p)}}{p \wedge \Box(p \to \circ p) \to \circ p \wedge \circ\Box(p \to \circ p)}}{p \wedge \Box(p \to \circ p) \to \circ(p \wedge \Box(p \to \circ p))}}{p \wedge \Box(p \to \circ p) \to p \wedge \circ(p \wedge \Box(p \to \circ p))}}{p \wedge \Box(p \to \circ p) \to \Box p}$$

Fig. 6. Greatest fixpoint reasoning: proof tree of $\vdash p \wedge \Box(p \to \circ p) \to \Box p$

The reader will notice that both this and the next subsections are short. Most texts are about helping the reader understand how the example proof goals are set up and *not* how they are proved. This is exactly the point: their proofs are not different from the proofs we have seen in Sections 4.3.1-4.3.3 for separation logic, thanks to the generality of our proof rules.

Consider the following induction proof rule in the sound and complete LTL proof system: $\vdash p \wedge \Box(p \to \circ p) \to \Box p$. As defined in Section 3.4.2, the "always $\Box$" temporal operator is a greatest fixpoint: $\Box\varphi =_{\text{gfp}} \varphi \wedge \circ\Box\varphi$. To reason about it, we need a set of proof rules dual to those in Fig. 2, where the key rule, (GFP) (dual to (LFP)), is shown below:

$$(\textsc{gfp})\quad \frac{\varphi \to \psi_i[\varphi/q]}{\varphi \to q(\tilde{y})}\quad q(\tilde{y}) =_{\text{gfp}} \bigvee \psi_i$$

(GFP) is used to discharge the RHS $\Box p$ of the proof goal. We show the self-explanatory proof tree in Fig. 6. Note that during the proof we use the distributivity law provided by the ML theory Γ^{LTL} in Section 3.4.2, denoted as proof step ($\circ\wedge$) in Fig. 6.

4.3.5 A Program Verification Example from Reachability Logic (RL). We have discussed RL and showed its ML theory in Section 3.4.3. 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. 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.

```
sum ≡ 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:

$$\varphi_{pre} \equiv \left\langle \left\langle \texttt{sum} \right\rangle_{\texttt{code}} \left\langle \texttt{n}\mapsto n,\ \texttt{s}\mapsto s \right\rangle_{\texttt{state}} \right\rangle_{\texttt{cfg}} \ \wedge\ n \geq 1$$
$$\varphi_{post} \equiv \left\langle \left\langle \cdot \right\rangle_{\texttt{code}} \left\langle \texttt{n}\mapsto 0, \texttt{s}\mapsto s + n(n-1)/2 \right\rangle_{\texttt{state}} \right\rangle_{\texttt{cfg}}$$

$$
\begin{prooftree}
\AxiomC{$S \vdash \mathsf{True}$}
\RightLabel{SMT}
\UnaryInfC{$S \vdash \varphi'_3 \to \varphi'_3$}
\RightLabel{PM}
\UnaryInfC{$S \vdash \varphi'_3 \to \Diamond\psi$}
\RightLabel{FRAME}
\UnaryInfC{$S \vdash \bullet^3\varphi'_3 \to \bullet^3\Diamond\psi$}
\RightLabel{UNFOLD-R}
\UnaryInfC{$S \vdash \bullet^3\varphi'_3 \to \Diamond\psi$}
\RightLabel{FOL}
\UnaryInfC{$S \vdash \circ^4(\forall n_1 \forall s_1.\,(C \multimap \Diamond\psi)) \wedge \bullet^3\varphi'_3 \to \Diamond\psi$}
\RightLabel{APP-SYM}
\UnaryInfC{$S \cup \{\varphi_{pre} \to (\bullet^3\varphi'_3 \vee \bullet^4\varphi_4)\} \vdash \circ^4(\forall n_1 \forall s_1.\,(C \multimap \Diamond\psi)) \wedge \varphi_{pre} \to \Diamond\psi$}
\RightLabel{UNWRAP}
\UnaryInfC{$S \cup \{\varphi_{pre} \to (\bullet^3\varphi'_3 \vee \bullet^4\varphi_4)\} \vdash \circ^4(\forall n_1 \forall s_1.\,(C \multimap \Diamond\psi)) \to (C \multimap \Diamond\psi)$}
\RightLabel{ELIM-$\forall$}
\UnaryInfC{$S \cup \{\varphi_{pre} \to (\bullet^3\varphi'_3 \vee \bullet^4\varphi_4)\} \vdash \circ^4(\forall n_1 \forall s_1.\,(C \multimap \Diamond\psi)) \to \forall n_1 \forall s_1.\,(C \multimap \Diamond\psi)$}
\RightLabel{LFP}
\UnaryInfC{$S \cup \{\varphi_{pre} \to (\bullet^3\varphi'_3 \vee \bullet^4\varphi_4)\} \vdash \mu f.\,\circ^4 f \to \forall n_1 \forall s_1.\,(C \multimap \Diamond\psi)$}
\RightLabel{INTRO-$\forall$}
\UnaryInfC{$S \cup \{\varphi_{pre} \to (\bullet^3\varphi'_3 \vee \bullet^4\varphi_4)\} \vdash \mu f.\,\circ^4 f \to (C \multimap \Diamond\psi)$}
\RightLabel{WRAP}
\UnaryInfC{$S \cup \{\varphi_{pre} \to (\bullet^3\varphi'_3 \vee \bullet^4\varphi_4)\} \vdash \varphi_{pre} \to (\mu f.\,\circ^4 f \to \Diamond\psi)$}
\RightLabel{SYM}
\UnaryInfC{$S \cup \{\varphi_{pre} \to (\bullet^3\varphi'_3 \vee \bullet^3\varphi_3)\} \vdash \varphi_{pre} \to (\mu f.\,\circ^3 f \to \Diamond\psi)$}
\RightLabel{SYM}
\UnaryInfC{$S \cup \{\varphi_{pre} \to \bullet^2(\varphi'_2 \vee \varphi_2)\} \vdash \varphi_{pre} \to (\mu f.\,\circ^2 f \to \Diamond\psi)$}
\RightLabel{SYM}
\UnaryInfC{$S \cup \{\varphi_{pre} \to \bullet\varphi_1\} \vdash \varphi_{pre} \to (\mu f.\,\circ f \to \Diamond\psi)$}
\RightLabel{SYM}
\UnaryInfC{$S \cup \{\varphi_{pre} \to \varphi_{pre}\} \vdash \varphi_{pre} \to (\mu f.\,\circ f \to \Diamond\psi)$}
\RightLabel{REACH}
\UnaryInfC{$S \cup \{\varphi_{pre} \to \varphi_{pre}\} \vdash \varphi_{pre} \Rightarrow \psi$}
\end{prooftree}
$$

$$
\begin{prooftree}
\AxiomC{$S \vdash \mathsf{True}$}
\RightLabel{SMT}
\UnaryInfC{$S \vdash \mathsf{SUM}(1, n'_1, s'_1, s_2) \to s'_1 = s_1 + n'_1 \wedge n'_1 = n_1 - 1 \wedge \mathsf{SUM}(1, n'_1, s'_1, s_2)$}
\RightLabel{PM}
\UnaryInfC{$S \vdash \mathsf{SUM}(1, n'_1, s'_1, s_2) \to \exists x \exists y.\, y = s_1 + x \wedge x = n_1 - 1 \wedge \mathsf{SUM}(1, x, y, s_2)$}
\RightLabel{UNFOLD-R}
\UnaryInfC{$S \vdash \mathsf{SUM}(1, n'_1, s'_1, s_2) \to \mathsf{SUM}(1, n_1, s_1, s_2)$}
\RightLabel{PM}
\UnaryInfC{$S \vdash \Diamond\psi' \to \Diamond\psi$}
\RightLabel{MATCH-CTX}
\UnaryInfC{$S \vdash \forall n_1 \forall s_1.\,(C \multimap \Diamond\psi) \wedge \varphi_4 \to \Diamond\psi$}
\RightLabel{FRAME}
\UnaryInfC{$S \vdash \bullet^4(\forall n_1 \forall s_1.\,(C \multimap \Diamond\psi) \wedge \varphi_4) \to \bullet^4\Diamond\psi$}
\RightLabel{UNFOLD-R}
\UnaryInfC{$S \vdash \bullet^4(\forall n_1 \forall s_1.\,(C \multimap \Diamond\psi) \wedge \varphi_4) \to \Diamond\psi$}
\RightLabel{$\circ\bullet$}
\UnaryInfC{$S \vdash \circ^4(\forall n_1 \forall s_1.\,(C \multimap \Diamond\psi)) \wedge \bullet^4\varphi_4 \to \Diamond\psi$}
\end{prooftree}
$$

$$
\begin{aligned}
&\mathsf{SUM}(l, u, b, s) =_{\mathsf{lfp}} (l > u \wedge s = b) \vee (\exists b_1 \exists u_1.\, b_1 = b + u_1 \wedge u_1 = u - 1 \wedge \mathsf{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}} \wedge n_2 = 0 \wedge \mathsf{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}} \wedge n_2 = 0 \wedge \mathsf{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}} \wedge 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}} \wedge 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}} \wedge 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}} \wedge n_1 \geq 1 && C[h] \equiv \varphi_{pre} \wedge 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}} \wedge 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}} \wedge n_1 \geq 1 && s'_1 \equiv s_1 + n_1 - 1
\end{aligned}
$$

Fig. 7. Verifying functional correctness of sum in terms of reachability rules

Following RL convention, we write configurations in *cells* such as $\langle \ldots \rangle_{\mathtt{code}}$, $\langle \ldots \rangle_{\mathtt{state}}$; 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 dot " $\cdot$ ", 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 3.4.3, $\varphi_{pre} \Rightarrow \varphi_{post}$ is a notation for $\varphi_{pre} \to (\mathsf{WF} \to \Diamond\varphi_{post})$, where $\mathsf{WF} = \mu X.\, \circ X$ is matched by all well-founded configurations (i.e., those without infinite execution traces) and $\Diamond\varphi_{post} = \mu X.\, \varphi_{\mathrm{post}} \vee \bullet X$ 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 Fig. 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 $S \cup S_i \vdash \varphi_i \rightarrow \psi_i$ 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 $S \cup \{\varphi_{pre} \rightarrow \varphi_{pre}\} \vdash \varphi_{pre} \rightarrow \Diamond_w \psi$, 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 $\varphi_{pre} \rightarrow \varphi_{pre}$ 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:

$$(\textsc{sym})\ \frac{S \cup S_k \vdash \varphi \rightarrow (\mu f.\ \circ^j f \rightarrow \Diamond\psi)}{S \cup \{\varphi \rightarrow \varphi'\} \vdash \varphi \rightarrow (\mu f.\ \circ^i f \rightarrow \Diamond\psi)}\quad \text{if } S_k \neq \emptyset \wedge i \geq 1 \text{ where } (S_k, j) = \textsc{next}(\varphi')$$

where NEXT takes the current symbolic configuration, executes it according to the semantics S, and outputs a rule that specifies the step (implemented similarly to [Ştefănescu et al. 2016]) and the number of steps taken. We stop execution when the code cell $\langle \ldots \rangle_{\texttt{code}}$ 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. $\{\varphi_{pre} \rightarrow (\bullet^3 \varphi_3' \vee \bullet^4 \varphi_4)\}$) will be used to simplify φ_{pre} later (e.g. as in the application of (APP-SYM)).

5 ALGORITHMS

As shown in Fig. 1, our generic ML prover runs a simple depth-first proof search (DFS) algorithm on top of the proof rules in Fig. 2. In this section, we show the top-level DFS algorithm in Fig. 8a. We also show the pattern matching algorithms used by the proof rules (PM) and (MATCH-CTX).

5.1 Top-Level Proof Search Algorithm

The top-level proof search algorithm in Fig. 8a starts with procedure Prove on the goal $\vdash \varphi \rightarrow \psi$, which uses two counters $c_{\text{RU}}, c_{\text{KT}}$, both initialized to zero, to keep track of how many times (UNFOLD-R) and (KT) have been applied. Proof search terminates (unsuccessfully) if they exceed the preset search bounds, so the proof procedure is incomplete, which is expected. Specifically, the algorithm consists of the following two cases:

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.

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 LHS, the proof goal is trivially true, which is denoted trivially_true in Fig. 8a, and is removed immediately.

Fig. 8a 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

```
function Prove(φ → ψ, c_RU, c_KT)
⟨1⟩  if (BasicProof(φ → ψ)) return true
⟨2⟩  let {p_i}:=rec_sym(φ), {q_i}:=rec_sym(ψ), OrSet := ∅
⟨3⟩  foreach (∀ỹ. C′ ⊸ φ′) ∈ φ    /* (MATCH-CTX)*/
⟨4⟩    C_0:=φ \ {∀ỹ. C′ ⊸ φ′}
⟨5⟩    (C_rest, θ):=cm(C_0, C′, ỹ)    /*Section 5.2.1*/
⟨6⟩    φ′′:=C_rest ∪ φ′θ
⟨7⟩    ob:=[φ′′ → ψ, c_RU, c_KT], OrSet ∪={{ob}}
⟨8⟩  foreach p_i ∈ φ, q_i′ ∈ ψ    /* (FRAME)*/
⟨9⟩    if (θ := pm([p_i], [q_i′], FV(ψ) \ FV(φ))) ≠ Failure
⟨10⟩     φ′:=φ \ {p_i}, ψ′:=(ψ \ {q_i′})θ
⟨11⟩     ob:=[φ′ → ψ′, c_RU, c_KT], OrSet ∪={{ob}}
⟨12⟩ if (c_RU < MAXRIGHTBOUND)    /* (UNFOLD-R)*/
⟨13⟩   foreach (q_i ∈ ψ)
⟨14⟩     foreach (ψ_j ∈ ({ψ_1 ... ψ_k} := UNFOLD(ψ, q_i)))
⟨15⟩       ob := [φ → ψ_j, c_RU + 1, c_KT], OrSet ∪={{ob}}
⟨16⟩ if (c_KT < KTBOUND)    /* (KT) */
⟨17⟩   foreach (p_i ∈ φ)
⟨18⟩     foreach (φ_j ∈ ({φ_1, φ_2, ... φ_l} := KT(φ, p_i)))
⟨19⟩       ob := [φ_j → ψ, c_RU, c_KT + 1]
⟨20⟩       if (trivially_true(ob)) continue
⟨21⟩       Obs := Obs ∪ {ob}
⟨22⟩     OrSet ∪= {Obs}
⟨23⟩ if (OrSet = ∅) return false
⟨24⟩ OrSet := OrderByHeuristics(OrSet)
⟨25⟩ foreach (Obs ∈ OrSet)
⟨26⟩   if (ProveAll(Obs)) return true
⟨27⟩ return false
endfunction
```

(a) Top-Level Proof Search Algorithm

```
function ProveAll(Obs)
⟨1⟩  foreach ([φ → ψ, c_RU, c_KT] ∈ Obs)
⟨2⟩    if (not Prove(φ → ψ, c_RU, c_KT))
⟨3⟩   return false;
⟨4⟩  return true
endfunction

function pm([ψ_i]_1^m, [φ_i]_1^m, Vs)
⟨5⟩  if m = 0 return { }
⟨6⟩  if ψ_1 ≡ σ(ψ̃_1) and φ_1 ≡ σ′(φ̃_1)
⟨7⟩    if σ ≠ σ′ return Failure
⟨8⟩    else if length(ψ̃_1) ≠ length(φ̃_1)
⟨9⟩      return Failure
⟨10⟩   else
⟨11⟩     [ψ_i′]_1^m′ = ψ̃_1 ∪ [ψ_i]_1^m
⟨12⟩     [φ_i′]_1^m′ = φ̃_1 ∪ [φ_i]_1^m
⟨13⟩     return pm([ψ_i′]_1^m′, [φ_i′]_1^m′, Vs)
⟨14⟩ if ψ_1 ≡ x and x ∉ Vs
⟨15⟩   if φ_1 ≡ x
⟨16⟩     return pm([ψ_i]_2^m, [φ_i]_2^m, Vs)
⟨17⟩   else
⟨18⟩     return Failure
⟨19⟩ if ψ_1 ≡ x and x ∈ Vs
⟨20⟩   [ψ_i′]_2^m = [ψ_i]_2^m{x ↦ φ_1}
⟨21⟩   [φ_i′]_2^m = [φ_i]_2^m{x ↦ φ_1}
⟨22⟩   θ′ = pm([ψ_i′]_2^m, [φ_i′]_2^m, Vs)
⟨23⟩   return {x ↦ φ_1} ∪ θ′
⟨24⟩ return Failure
endfunction
```

(b) Pattern Matching Algorithm

Fig. 8. Implementation and Algorithms of the Proof Rules in Fig. 2

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 RHS are prioritized.
(2) Goals with recursive patterns on the RHS but not on the LHS are considered next.
(3) Goals with fewer existential variables are prioritized.

5.2 Matching Algorithms

As discussed in Section 4.2, our proof framework is parametric in two matching algorithms: cm used by (MATCH-CTX) and pm used by (PM). We discuss these two algorithms below.

5.2.1 Context Matching. 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:

$$\vdash C_{outer}[\forall \tilde{y}.\, (C' \multimap \varphi')] \to \psi \tag{5}$$

$\mathsf{cm}(C_{outer}, C', \tilde{y})$ 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:

$$\vdash C_{outer} \rightarrow \exists \tilde{y}.\, C' \tag{6}$$

In (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). The change of the quantifier $\tilde{y}$ from universal to existential is because we have moved the inner context C' from LHS to RHS.

To deal with (6), cm will call the modified version of the Prove function in Figure 8a. 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 LHS , denoted C_{rest}, after consuming all the matched constraints from the RHS, 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), we have $\vdash C_{outer} \rightarrow C'\theta$. Furthermore, C_{rest} is the remaining part after removing the constraint of $C'\theta$ from C_{outer}, so we have $C_{rest}[C'\theta[C'\theta \multimap \varphi'\theta]] \rightarrow \psi$. As a result, we now can proceed to prove new proof goal $C_{rest}[\varphi'\theta] \rightarrow \psi$.

5.2.2 Pattern Matching. Procedure pm, used by rule (PM), implements a naive, brute-force algorithm as shown in Fig. 8b 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 Vs with $Vs \cap \bigcup_1^n \mathrm{FV}(\varphi_j) = \emptyset$ and $Vs \subseteq \bigcup_1^m \mathrm{FV}(\psi_i)$.

Then it returns Failure or the match result θ with $\mathsf{domains}(\theta) \subseteq Vs$ and $\psi_i\theta \equiv \varphi_i$, for all i.

6 EVALUATION

We implemented our proof framework in the $\mathbb{K}$ framework (http://kframework.org). $\mathbb{K}$ 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 $\mathbb{K}$.

As discussed in Section 1, 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 [Sighireanu et al. 2019]. 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., [Goldblatt 1992; Lichtenstein and Pnueli 2000]). 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 (Section 3.4.3).

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.

Table 1. Selected separation logic properties, automatically proved by our prover

$\mathsf{sorted\_list}(x, min) \rightarrow \mathsf{list}(x)$
$\mathsf{sorted\_list}_1(x, len, min) \rightarrow \mathsf{list}_1(x, len)$
$\mathsf{sorted\_list}_1(x, len, min) \rightarrow \mathsf{sorted\_list}(x, min)$
$\mathsf{sorted\_ls}(x, y, min, max) * \mathsf{sorted\_list}(y, min_2) \wedge max \leq min_2 \rightarrow \mathsf{sorted\_list}(x, min)$
$\mathsf{lr}(x, y) * \mathsf{list}(y) \rightarrow \mathsf{list}(x)$
$\mathsf{lr}(x, y) \rightarrow \mathsf{ll}(x, y)$
$\mathsf{ll}(x, y) \rightarrow \mathsf{lr}(x, y)$
$\mathsf{ll}_1(x, y, len_1) * \mathsf{ll}_1(y, z, len_2) \rightarrow \mathsf{ll}_1(x, z, len_1{+}len_2)$
$\mathsf{lr}_1(x, y, len_1) * \mathsf{list}_1(y, len_2) \rightarrow \mathsf{list}_1(x, len_1{+}len_2)$
$\mathsf{ll}_1(x, last, len) * (last \mapsto new) \rightarrow \mathsf{ll}_1(x, new, len + 1)$
$\mathsf{dls}(x, y) * \mathsf{dlist}(y) \rightarrow \mathsf{dlist}(x)$
$\widehat{\mathsf{dls}}_1(x, y, len_1) * \widehat{\mathsf{dls}}_1(y, z, len_2) \rightarrow \widehat{\mathsf{dls}}_1(x, z, len_1{+}len_2)$
$\mathsf{dls}_1(x, y, len_1) * \mathsf{dlist}_1(y, len_2) \rightarrow \mathsf{dlist}_1(x, len_1{+}len_2)$
$\mathsf{avl}(x, hgt, min, max, balance) \rightarrow \mathsf{bstree}(x, hgt, min, max)$
$\mathsf{bstree}(x, height, min, max) \rightarrow \mathsf{bintree}(x, height)$

Our first evaluation is based on the standard separation logic benchmark set collected by SL-COMP'19 [Sighireanu et al. 2019]. 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 3.4.1. Note that the associativity and commutativity of $\varphi_1 * \varphi_2$ are handled by the *built-in* pattern matching algorithms (see Fig. 8b), 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 [Enea et al. 2017; Iosif et al. 2013]. 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 for SL. Compared with generic provers such as CYCLIST [Brotherston et al. 2012], our prover proves 13 more tests.

Table 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

Table 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)	$\varphi \wedge \Box(\varphi \to \circ\varphi) \to \Box\varphi$
(U_1)	$\varphi_1 \; U \; \varphi_2 \to \Diamond\varphi_2$
$(\text{U}_2.1)$	$\varphi_1 \; U \; \varphi_2 \to \varphi_2 \vee \varphi_1 \wedge \circ(\varphi_1 \; U \; \varphi_2)$
$(\text{U}_2.2)$	$\varphi_2 \vee \varphi_1 \wedge \circ(\varphi_1 \; U \; \varphi_2) \to \varphi_1 \; U \; \varphi_2$

sorted list is also a list, a typical verification condition arising in formal verification. Table 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 2, whereas the proof tree of the most interesting of them, (IND), has been given in Section 4.3.4. 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 4.3.5. 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). Following a similar idea, we also considered the prototypical heap-manipulating program `reverse` that reverses a singly-linked list, whose complete proof tree we exile to [Chen et al. 2020b].

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.

In summary, we evaluated our generic proof framework using four different logical systems and demonstrated its generality with respect to fixpoint reasoning. We find it encouraging that our generic framework is comparable in terms of automation with specialized state-of-the-art inductive provers for SL, while at the same time also works within other, distinct domains, such as LTL and program verification of partial correctness. We have noticed that one major bottleneck of our implementation is its relatively weak support for non-fixpoint reasoning, such as pattern matching and standard FOL reasoning. In the near future, we plan to improve the non-fixpoint reasoning of our prover, which shall increase its effectiveness at domain reasoning, and not only. We also plan to investigate smarter proof search strategies, which shall improve its efficiency and performance.

7 CONCLUSION

We proposed a unified proof framework for automated fixpoint reasoning based on matching logic, which allows us to encode formulae in other logical systems almost verbatim and reason about them using one generic but fixed set of proof rules. We explained why the existing proof system of matching logic is too low-level and not suitable for automated reasoning, and then we proposed a new proof framework containing higher-level proof rules and proved its soundness. We demonstrated the generality and effectiveness of our proof framework using four distinct logical

systems from first-order logic with least fixpoints and separation logic to program verification. Our experimental results show that our generic proof framework is competitive among specialized provers and algorithms. We hope our work presented in this paper brings some evidence that a unified proof framework for automated inductive formal reasoning *is possible*.

ACKNOWLEDGMENTS

We warmly thank the anonymous OOPSLA reviewers and our shepherd. Their wit and dedication has helped us improve the presentation. This work was supported in part by NSF CNS 16-19275. This material is based upon work supported by the United States Air Force and DARPA under Contract No. FA8750-18-C-0092. This research is also partly supported by the National Research Foundation, Prime Minister's Office, Singapore under its Campus for Research Excellence and Technological Enterprise (CREATE) programme.

REFERENCES

David Baelde, Dale Miller, and Zachary Snow. 2010. Focused inductive theorem proving. In *Proceedings of the 5th International Joint Conference on Automated Reasoning (IJCAR'10)*. Springer, Edinburgh, UK, 278–292. https://doi.org/10.1007/978-3-642-14203-1_24

Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)*. Springer, Berlin, Heidelberg, 171–177.

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2004. A decidable fragment of separation logic. In *Proceedings of the 24th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'04)*. Springer, Heidelberg, Germany, 97–109. https://doi.org/10.1007/978-3-540-30538-5_9

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2005. Symbolic execution with separation logic. In *Proceedings of the 3rd Asian conference on Programming Languages and Systems (APLAS'05)*. Springer, Tsukuba, Japan, 52–68. https://doi.org/10.1007/11575467_5

Nikolaj Bjørner and Joe Hendrix. 2009. Linear functional fixed-points. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV'09)*. Springer, Grenoble, France, 124–139. https://doi.org/10.1007/978-3-642-02658-4_13

Patrick Blackburn, Maarten de Rijke, and Yde Venema. 2001. *Modal logic*. Cambridge University Press, New York, NY, USA.

Denis Bogdănaş and Grigore Roşu. 2015. K-Java: A complete semantics of Java. In *Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL'15)*. ACM, Mumbai, India, 445–456. https://doi.org/10.1145/2676726.2676982

William W. Boone. 1958. The word problem. *Proceedings of the National Academy of Sciences* 44, 10 (1958), 1061–1065. https://doi.org/10.1073/pnas.44.10.1061

Ahmed Bouajjani, Cezara Drăgoi, Constantin Enea, and Mihaela Sighireanu. 2009. A logic-based framework for reasoning about composite data structures. In *Proceedings of the 20th International Conference on Concurrency Theory (CONCUR'09)*. Springer, Bologna, Italy, 178–195. https://doi.org/10.1007/978-3-642-04081-8_13

James Brotherston, Dino Distefano, and Rasmus Lerchedahl Petersen. 2011. Automated cyclic entailment proofs in separation logic. In *Proceedings of the 23rd International Conference on Automated Deduction (CAV'11)*. Springer, Utah, USA, 131–146.

James Brotherston, Carsten Fuhs, Juan A. Navarro Pérez, and Nikos Gorogiannis. 2014. A Decision Procedure for Satisfiability in Separation Logic with Inductive Predicates. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)* (Vienna, Austria) *(CSL-LICS '14)*. ACM, New York, NY, USA, Article 25, 10 pages. https://doi.org/10.1145/2603088.2603091

James Brotherston, Nikos Gorogiannis, and Rasmus L. Petersen. 2012. A generic cyclic theorem prover. In *Programming Languages and Systems*, Ranjit Jhala and Atsushi Igarashi (Eds.). Springer, Kyoto, Japan, 350–367.

James Brotherston and Max Kanovich. 2014. Undecidability of propositional separation logic and its neighbours. *J. ACM* 61, 2, Article 14 (April 2014), 43 pages. https://doi.org/10.1145/2542667

Xiaohong Chen, Dorel Lucanu, and Grigore Roşu. 2020a. *Initial algebra semantics in matching logic*. Technical Report http://hdl.handle.net/2142/107781. University of Illinois at Urbana-Champaign.

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

Xiaohong Chen and Grigore Roşu. 2020. A general approach to define binders using matching logic. In *Proceedings of the 25th ACM SIGPLAN International Conference on Functional Programming (ICFP'20)*. ACM/IEEE.

Xiaohong Chen, Minh-Thai Trinh, Nishant Rodrigues, Lucas Peña, and Grigore Roşu. 2020b. *Towards a unified proof framework for automated fixpoint reasoning using matching logic*. Technical Report. University of Illinois at Urbana-Champaign. http://hdl.handle.net/2142/108369

Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. 2012. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Science of Computer Programming* 77, 9 (Aug. 2012), 1006–1036. https://doi.org/10.1016/j.scico.2010.07.004

Duc-Hiep Chu, Joxan Jaffar, and Minh-Thai Trinh. 2015. 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, Portland Oregon, 457–466. https://doi.org/10.1145/2737924.2737984

Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 2009. VCC: A practical system for verifying concurrent C. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs'09)*. Springer, Munich, Germany, 23–42. https://doi.org/10.1007/978-3-642-03359-9

Andrei Ştefănescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore Roşu. 2016. 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, Amsterdam, The Netherlands, 74–91. https://doi.org/10.1145/2983990.2984027

Leonardo De Moura and Nikolaj Bjørner. 2008. 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, Budapest, Hungary, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Constantin Enea, Ondřej Lengál, Mihaela Sighireanu, and Tomáš Vojnar. 2017. Compositional entailment checking for a fragment of separation logic. *Formal Methods in System Design* 51, 3 (Dec. 2017), 575–607. https://doi.org/10.1007/s10703-017-0289-4

Zoltán Ésik. 1997. Completeness of Park induction. *Theoretical Computer Science* 177, 1 (1997), 217–283. https://doi.org/10.1016/S0304-3975(96)00240-X

Robert Goldblatt. 1992. *Logics of Time and Computation* (2. ed.). Number 7 in CSLI Lecture Notes. Center for the Study of Language and Information, Stanford, CA.

Yuri Gurevich and Saharon Shelah. 1985. Fixed-point extensions of first-order logic. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science (SFCS'85)*. IEEE, Portland, OR, 346–353.

Chris Hathhorn, Chucky Ellison, and Grigore Roşu. 2015. Defining the undefinedness of C. In *Proceedings of the 36th annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*. ACM, Portland Oregon, 336–345. https://doi.org/10.1145/2813885.2737979

Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, Brandon Moore, Yi Zhang, Daejun Park, Andrei Ştefănescu, and Grigore Roşu. 2018. KEVM: A complete semantics of the Ethereum virtual machine. In *Proceedings of the 2018 IEEE Computer Security Foundations Symposium (CSF'18)*. IEEE, Oxford, UK, 204–217. http://jellopaper.org.

C. A. R. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580. https://doi.org/10.1145/363235.363259

Gerard J. Holzmann. 1997. The model checker SPIN. *IEEE Trans. Softw. Eng.* 23, 5 (1997), 279–295. https://doi.org/10.1109/32.588521

Radu Iosif, Adam Rogalewicz, and Jiri Simacek. 2013. The tree width of separation logic with recursive definitions. In *Proceedings of the 24th International Conference on Automated Deduction (CADE'13)*. Springer, New York, USA, 21–38. https://doi.org/10.1007/978-3-642-38574-2_2

Bart Jacobs, Jan Smans, and Frank Piessens. 2010. A quick tour of the VeriFast program verifier. In *Proceedings of the 8th Asian Symposium of Programming Languages and Systems (APLAS'10)*. Springer, Shanghai, China, 304–311. https://doi.org/10.1007/978-3-642-17164-2

Jens Katelaan, Christoph Matheja, and Florian Zuleger. 2019. Effective Entailment Checking for Separation Logic with Inductive Definitions. In *Tools and Algorithms for the Construction and Analysis of Systems*, Tomáš Vojnar and Lijun Zhang (Eds.). Springer International Publishing, Cham, 319–336.

Laura Kovács, Simon Robillard, and Andrei Voronkov. 2017. Coming to terms with quantified reasoning. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 260–270. https://doi.org/10.1145/3009837.3009887

Dexter Kozen. 1982. Results on the propositional μ-calculus. In *Proceedings of the 9th Colloquium on Automata, Languages and Programming*. Springer, Ninth Colloquium Aarhus, Denmark, 348–359. https://doi.org/10.1007/BFb0012782

Shuvendu Lahiri and Shaz Qadeer. 2008. 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, California USA, 171–182. https://doi.org/10.1145/1328438.1328461

K. Rustan M. Leino and Michał Moskal. 2014. Co-induction simply. In *Proceedings of the 19th International Symposium on Formal Methods (FM'14)*. Springer, Singapore, 382–398. https://doi.org/10.1007/978-3-319-06410-9

Orna Lichtenstein and Amir Pnueli. 2000. Propositional Temporal Logics: Decidability and Completeness. *Logic Journal of the IGPL* 8, 1 (2000), 55–85. http://dblp.uni-trier.de/db/journals/igpl/igpl8.html#LichtensteinP00

Christof Löding, Madhusudan Parthasarathy, and Lucas Peña. 2017. Foundations for natural proofs and quantifier instantiation. *Proceedings of the ACM on Programming Languages (POPL'17)* 2, 1 (2017), 1–30. https://doi.org/10.1145/3158098

Dorel Lucanu and Grigore Roşu. 2007. CIRC: A circular coinductive prover. In *Proceedings of the 2nd international conference on Algebra and coalgebra in computer science (CALCO'07)*. Springer, Berlin, Heidelberg, Bergen, Norway, 372–378.

The Coq development team. 2004. *The Coq proof assistant reference manual*. LogiCal Project.

Daejun Park, Andrei Ştefănescu, and Grigore Roşu. 2015. KJS: A complete formal semantics of JavaScript. In *Proceedings of the 36th annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*. ACM, Portland Oregon, 346–356. https://doi.org/10.1145/2737924.2737991

Juan Antonio Navarro Pérez and Andrey Rybalchenko. 2011. 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, California, USA, 556–566. https://doi.org/10.1145/1993498.1993563

Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2013. Automating separation logic using SMT. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV'13)*. Springer, Saint Petersburg, Russia, 773–789. https://doi.org/10.1007/978-3-642-39799-8_54

Amir Pnueli. 1977. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FCS'77)*. IEEE, IEEE, DC, USA, 46–57.

Zvonimir Rakamarić, Jesse Bingham, and Alan J. Hu. 2007a. 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)*. Springer, California, USA, 106–121. https://doi.org/10.1007/978-3-540-69738-1_8

Zvonimir Rakamarić, Roberto Bruttomesso, Alan J. Hu, and Alessandro Cimatti. 2007b. Verifying heap-manipulating programs in an SMT framework. In *Proceedings of the 5th International Symposium on Automated Technology for Verification and Analysis (ATVA'07)*. Springer, Tokyo, Japan, 237–252. https://doi.org/10.1007/978-3-540-75596-8_18

Silvio Ranise and Calogero Zarba. 2006. 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, Macao, China, 206–215. https://doi.org/10.1109/sefm.2006.7

John C. Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS'02)*. IEEE, Copenhagen, Denmark, 55–74. https://doi.org/10.1109/lics.2002.1029817

Grigore Roşu. 2017. Matching logic. *Logical Methods in Computer Science* 13, 4 (Dec. 2017), 1–61. https://doi.org/10.23638/lmcs-13(4:28)2017

Grigore Roşu, Andrei Ştefănescu, Ştefan Ciobâcă, and Brandon M. Moore. 2013. One-path reachability logic. In *Proceedings of the 28th Symposium on Logic in Computer Science (LICS'13)*. IEEE, New Orleans, USA, 358–367. https://doi.org/10.1109/lics.2013.42

Mihaela Sighireanu, Juan A. Navarro Pérez, Andrey Rybalchenko, Nikos Gorogiannis, Radu Iosif, Andrew Reynolds, Cristina Serban, Jens Katelaan, Christoph Matheja, Thomas Noll, Florian Zuleger, Wei-Ngan Chin, Quang Loc Le, Quang-Trung Ta, Ton-Chanh Le, Thanh-Toan Nguyen, Siau-Cheng Khoo, Michal Cyprian, Adam Rogalewicz, Tomas Vojnar, Constantin Enea, Ondrej Lengal, Chong Gao, and Zhilin Wu. 2019. SL-COMP: Competition of solvers for separation logic. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen (Eds.). Springer International Publishing, Cham, 116–132.

Quang-Trung Ta, Ton Chanh Le, Siau-Cheng Khoo, and Wei-Ngan Chin. 2019. Automated mutual induction proof in separation logic. *Formal Aspects of Computing* 31, 2 (April 2019), 207–230. https://doi.org/10.1007/s00165-018-0471-5

Alfred Tarski. 1955. A lattice-theoretical fixpoint theorem and its applications. *Pacific journal of Mathematics* 5, 2 (1955), 285–309. https://doi.org/10.2140/pjm.1955.5.285

The Isabelle development team. 2018. Isabelle. https://isabelle.in.tum.de/.

Hiroshi Unno, Sho Torii, and Hiroki Sakamoto. 2017. Automating induction for solving Horn clauses. In *Proceedings of the 29th International Conference on Computer Aided Verification (CAV'17)*. Springer, Heidelberg, Germany, 571–591. https://doi.org/10.1007/978-3-319-63390-9_30