Technical Report
The Deduction System of Matching Logic

Formal Systems Laboratory[1]

[1]University of Illinois, Urbana-Champaign, USA

August 21, 2017

Abstract

This paper proposes a sound and complete deductive system of matching logic, proves dozens of useful metatheorems about the deductive system, and shows its applications in (1) modeling transition systems; (2) symbolic executions; and (3) reasoning with contexts.

1 Syntax

Formulas of matching logic, called *patterns*, are written in a formal language, denoted as $\mathcal{L}$, whose grammar is listed in (1). The language $\mathcal{L}$ is many-sorted. A signature of $\mathcal{L}$ contains not only a finite set Σ of *symbols*, but also a finite nonempyty set S of *sorts*. Each symbol $\sigma \in \Sigma$ is, of course, sorted, with a fixed nonempty arity. We write $\sigma \in \Sigma_{s_1,\ldots,s_n,s}$ to emphasize that σ takes n arguments (with argument sorts $s1, \ldots, s_n$) and returns a pattern in sort s, but we hope in most cases sorting is clear from context.

The grammar for $\mathcal{L}$, as defined below, is almost identical to first-order logic, except that in $\mathcal{L}$ there is no difference between relational (predicate) and functional symbols, and we accept first-order terms as patterns in matching logic.

$$\begin{aligned} P ::= {} & x \\ & \mid P_1 \wedge P_2 \\ & \mid \neg P \\ & \mid \forall x.P \\ & \mid \sigma(P_1, \ldots, P_n), \end{aligned} \tag{1}$$

where the universal quantifier ($\forall x$) behaves as a binder with alpha-renaming always assumed.

For simplicity, we did not mention sorting in the grammar definition, and assume it should be clear to all readers. For example, in $P_1 \wedge P_2$, both patterns P_1 and P_2 should have the same sort, and that sort is the sort of $P_1 \wedge P_2$. The sort of $\forall x.P$ is the sort of

P, while the sort of variable x does not matter. To see why it is the case, consider the pattern $\exists x.list(x, 1 \cdot 3 \cdot 5)$, which is the set of all memory configurations that has a list $(1, 3, 5)$ in it.

Propositional connectives are always assumed, including disjunction ($\vee$), implication ($\rightarrow$), and equivalence ($\leftrightarrow$). Existential quantifier ($\exists x$) is defined by universal quantifier ($\forall x$) in the normal way. The bottom pattern ($\bot_s$) and the top pattern ($\top_s$) in sort s are given by $x \wedge \neg x$ and $\neg \bot_s$, respectively, where x is a variable in sort s. It does not matter which variable we pick.

Definition 1. *The set of free variables in a pattern P is denoted as $freevars(P)$, defined recursively over the structure of P as usual. A pattern is said to be* closed *if $freevars(P) = \emptyset$. The universal (existential) generalization of P, denoted as $\forall P(\exists P)$, is defined as $\forall x_1 \ldots \forall x_n.P$ ($\exists x_1 \ldots \exists x_n.P$) where $freevars(P) = \{x_1, \ldots, x_n\}$.*

1.1 Extended syntax

The formal language $\mathcal{L}$ is often extended with *definedness* symbols. For s_1, s_2 are two sorts, the definedness symbol $\lceil \_ \rceil^{s_2}_{s_1} \in \Sigma_{s_1,s_2}$ is a unary symbol with one argument sort s_1 and the result sort s_2. For a pattern P who has sort s_1, the pattern $\lceil \_ \rceil^{s_2}_{s_1}(P)$ is often written as $\lceil P \rceil^{s_2}_{s_1}$, or simply $\lceil P \rceil$.

Definedness symbols carry specific intended semantics. For each definedness symbol $\lceil \_ \rceil^{s_2}_{s_1}$, we add the pattern $\lceil x \rceil^{s_2}_{s_1}$ as an axiom to the deductive system, where x is a variable who has sort s_1. It does not matter which variable we pick.

With definedness symbols, we extends the formal language $\mathcal{L}$ with

$$
\begin{aligned}
&\lfloor P \rfloor^{s_2}_{s_1} := \neg \lceil \neg P \rceil^{s_2}_{s_1} \\
&P_1 =^{s_2}_{s_1} P_2 := \lfloor P_1 \leftrightarrow P_2 \rfloor^{s_2}_{s_1} \\
&P_1 \neq^{s_2}_{s_1} P_2 := \neg (P_1 =^{s_2}_{s_1} P_2) \\
&P_1 \subseteq^{s_2}_{s_1} P_2 := \lfloor P_1 \rightarrow P_2 \rfloor^{s_2}_{s_1} \\
&x \in^{s_2}_{s_1} P := x \subseteq^{s_2}_{s_1} P.
\end{aligned}
$$

Remark 2. *To prevent writing tangled subscripts and superscripts that indicate sorts of variables and patterns all the time, we omit them as much as possible, unless there is a chance of confusing things. A statement with sorting subscripts and superscripts omitted is treated as (possibly many) statements with the omitting sorting subscripts and superscripts completed in all possible well-formed ways.*

2 Deductive System

A deductive system is a recursive set of patterns as *axioms* and a finite set of *inference rules*. The deductive system of matching logic that we introduce in this section has been proved *sound* and *complete*.

2.1 The deductive system

Axioms are given by the following axiom schemata where P, Q, R are arbitrary patterns and x, y are arbitrary logic variables.

- (K1) $P \to (Q \to P)$
- (K2) $(P \to (Q \to R)) \to ((P \to Q) \to (P \to R))$
- (K3) $(\neg P \to \neg Q) \to (Q \to P)$
- (K4) $\forall x.P \to P[y/x]$
- (K5) $\forall x.(P \to Q) \to (P \to \forall x.Q)$ if x does not occur free in P
- (K6) $P_1 = P_2 \to (Q[P_1/x] \to Q[P_2/x])$
- (Df) $\lceil x \rceil$
- (M1) $x \in y = (x = y)$
- (M2) $x \in P \wedge Q = (x \in P) \wedge (x \in Q)$
- (M3) $x \in \neg P = \neg(x \in P)$
- (M4) $x \in \forall y.P = \forall y.x \in P$ where x is distinct from y
- (M5) $x \in \sigma(\dots, P_i, \dots) = \exists y.y \in P_i \wedge x \in \sigma(\dots, y, \dots)$ where y occurs free in the left hand side of the equation.

Remark 3. *Substitution is denoted as $Q[P/x]$. Alpha-renaming is always assumed in order to avoid free variables capturing. Therefore, free variables in P are kept free after the substitution, i.e., $\mathit{freevars}(P) \subseteq \mathit{freevars}(Q[P/x])$.*

Inference rules include

- (Modus Ponens) From P and $P \to Q$, deduce Q.
- (Universal Generalization) From P, deduce $\forall x.P$.
- (Membership Introduction) From P, deduce $\forall x.(x \in P)$, where x does not occur free in P.
- (Membership Elimination) From $\forall x.(x \in P)$, deduce P, where x does not occur free in P.

Theorem 4. *The proof system is sound and complete.*

Proof. Refer to [RTA15]. □

2.2 Metatheorems of the deductive system

Writing formal proofs is never easy. Derivations are prone to be lengthy and boring. To ease such difficulty, we here in this section introduce a dozen of lemmas (i.e., metatheorems) of the deductive system. Metatheorems discover all kinds of properties of the deductive system, from the simplest "$\vdash P = P$" to the complexest deduction theorem and the framing rule.

Proposition 5 (Tautology)**.** *For any propositional tautology $\mathcal{A}(p_1, \ldots, p_n)$ where $p_1, \ldots, p_n$ are all propositional variables in $\mathcal{A}$, and for any patterns $P_1, \ldots, P_n$,*

$$\vdash \mathcal{A}(P_1, \ldots, P_n).$$

Proof. No proof. □

Corollary 6. $\vdash \top$.

Proof. By definition, $\top = \neg\bot = \neg(x \wedge \neg x)$, where x is a matching logic variable who has the same sort with $\top$. Let proposition $\mathcal{A} = \neg(p \wedge \neg p)$ with p is a propositional variable. Then $\mathcal{A}$ is a propositional tautology. By Proposition 5, $\top = \mathcal{A}[x/p]$ is derivable in the proof system, i.e., $\vdash \top$. □

Proposition 7. $\vdash P \to Q$ *implies* $P \vdash Q$.

Proof. The proof is a simple application of the Modus Ponens inference rule, as shown in the next derivation tree.

$$\dfrac{\dfrac{\cdot}{P \to Q} \quad \dfrac{\cdot}{P}}{Q}\ \text{(MP)}$$

□

Proposition 8 ($\vee$-Introduction)**.** $\vdash P$ *implies* $\vdash P \vee Q$.

Proof.

$$\dfrac{\dfrac{\cdot}{\vdash P} \quad \dfrac{\cdot}{\vdash P \to (\neg Q \to P)}\ \text{(K1)}}{\dfrac{\vdash \neg Q \to P}{\vdash P \vee Q}\ \text{(Sugar)}}\ \text{(MP)}$$

□

Corollary 9 ($\to$-Introduction)**.** $\vdash P$ *implies* $\vdash Q \to P$ *and* $\vdash \neg P \to Q$.

Remark 10. *In general, $\vdash P \vee Q$ does not implies $\vdash P$ or $\vdash Q$. For example, we have shown that $\vdash \top$, and $\top$ is, by definition, just sugar of $\neg\bot = \neg(x \wedge \neg x) = \neg x \vee x$. It is clearly wrong if we conclude $\vdash \neg x$ or $\vdash x$. From a semantic point of view, it is easy to understand: the union of two sets is the total set does not imply that one of them is the total set.*

Proposition 11 ($\wedge$-Introduction and Elimination)**.** $\vdash P$ *and* $\vdash Q$ *iff* $\vdash P \wedge Q$.

Proof. ($\Rightarrow$).

$$\dfrac{\dfrac{\cdot}{\vdash P} \quad \dfrac{\dfrac{\cdot}{\vdash Q} \quad \dfrac{\cdot}{\vdash Q \to P \to P \wedge Q}\ \text{(Taut)}}{\vdash P \to P \wedge Q}\ \text{(MP)}}{\vdash P \wedge Q}\ \text{(MP)}$$

($\Leftarrow$). Left as an exercise. □

Equalities plays an important role in matching logic. Axiom (K6) is very powerful even though it looks quite simple. It basically says that whenever one establishes that $P = Q$, then the two patterns are interchangeable everywhere in any patterns, as concluded in the next lemma.

Lemma 12. *If* $\vdash P_1 = P_2$ *and* $\vdash Q[P_1/x]$*, then* $\vdash Q[P_2/x]$.

Proof.

$$\dfrac{\dfrac{\dfrac{\cdot}{\vdash P_1 = P_2} \quad \dfrac{\cdot}{\vdash P_1 = P_2 \to (Q[P_1/x] \to Q[P_2/x])}\ \text{(K6)}}{\vdash Q[P_1/x] \to Q[P_2/x]}\ \text{(MP)} \quad \dfrac{\cdot}{\vdash Q[P_1/x]}}{\vdash Q[P_2/x]}\ \text{(MP)}$$

□

Remind that the equality "=" is not a built-in logic connective in matching logic. It is the syntactic sugar of $\neg\lceil\neg(P \leftrightarrow Q)\rceil$, where $\lceil\_\rceil$ is the definedness symbol that we introduced before. One may wonder how to establish such equalities in the deduction system. Indeed, the proof system says little about how to derive an equality. Most of its axioms (except (K6)) and rules are not even about equalities. That is why the next proposition is quite useful in practice. It helps one to establish an equality pattern.

Proposition 13. $\vdash P \leftrightarrow Q$ *iff* $\vdash P = Q$.

Proof. That the right hand side implies the left is easy, so we left the proof as an exercise to the readers. In the following, we only prove that the left implies the right. By definition, $P = Q$ is the syntactic sugar of $\neg\lceil\neg\,(P \leftrightarrow Q)\rceil$, so we have the following derivation.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\cdot}{\vdash P \leftrightarrow Q}}{\vdash \forall y.(y \in P \leftrightarrow Q)}\ \text{(}\in\text{-Intro)}}{\vdash y \in P \leftrightarrow Q}\ \text{(K4, MP)}}{\vdash \neg(x \in \lceil y\rceil) \vee (y \in P \leftrightarrow Q)}\ \text{(}\vee\text{-Intro)}}{\vdash \forall y.(\neg(x \in \lceil y\rceil) \vee (y \in P \leftrightarrow Q)}\ \text{(}\forall y\text{-Gen)}}{\vdash \neg\exists y.\,(x \in \lceil y\rceil \wedge y \in \neg\,(P \leftrightarrow Q))}\ \text{(Sugar, K3, M3)}}{\vdash \neg\,(x \in \lceil\neg\,(P \leftrightarrow Q)\rceil)}\ \text{(K6, M5)}}{\vdash x \in \neg\lceil\neg\,(P \leftrightarrow Q)\rceil}\ \text{(K6, M3)}}{\vdash \forall x.\,(x \in \neg\lceil\neg\,(P \leftrightarrow Q)\rceil)}\ \text{(}\forall x\text{-Gen)}}{\vdash \neg\lceil\neg\,(P \leftrightarrow Q)\rceil}\ \text{(}\in\text{-Intro)}$$

□

Remark 14. *We never say $P = Q$ and $P \leftrightarrow Q$ are logically equivalent. One will never derive $\vdash (P = Q) = (P \leftrightarrow Q)$ for any patterns P and Q in the deductive system from a consistent set of axioms.*

Corollary 15. *The following propositions hold for any pattern P.*

1. $\vdash P$ *iff* $\vdash P = \top$.
2. $\vdash \neg P$ *iff* $\vdash P = \bot$.
3. $\vdash (P \wedge \top) = P$.
4. $\vdash (P \wedge \bot) = \bot$.
5. $\vdash (P \vee \top) = \top$.
6. $\vdash (P \vee \bot) = P$.
7. $\vdash \forall x.\top = \top$.
8. $\vdash \forall x.\bot = \bot$.
9. $\vdash \exists x.\top = \top$.
10. $\vdash \exists x.\bot = \bot$.
11. $\vdash (x \in \top) = \top$.
12. $\vdash (x \in \bot) = \bot$.
13. $\vdash P = P$.

Proof. Left as exercises. Hint: use Proposition 13. □

Equalities are important, both semantically and syntactically. First recall the next proposition of the semantics of equalities.

Proposition 16. *If $\models P = Q$, then in any model M and evaluation ρ, $\bar{\rho}(P) = \bar{\rho}(Q)$ in M.*

Roughly speaking, if we establish $\models P = Q$, then it means that P and Q are interchangeable in any ways in any models. The same thing happen in a syntactic prospective, too, as we have seen in Proposition 12. But we can do more, as shown in the next proposition.

Proposition 17. *$\vdash P = Q$ implies $\vdash \forall x.P = \forall x.Q$ and $\vdash \exists x.P = \exists x.Q$, where x is an arbitraty variable that may or may not be free in P or Q.*

Proof. To be continued. □

Proposition 18 (Functional Substitution)**.** *$\vdash \exists y.(Q = y) \rightarrow (\forall x.P \rightarrow P[Q/x])$, if y occurs free in Q.*

Proof. To be continued. □

Proposition 19. $\vdash x \in \lceil y \rceil$.

Proof.

$$\begin{aligned} &\vdash x \in \lceil y \rceil \\ \text{if } &\vdash \forall x.(x \in \lceil y \rceil) && (K5,\ K6, \text{ and Modus Ponens}) \\ \text{iff } &\vdash \lceil y \rceil. \end{aligned}$$

□

Proposition 20. $\vdash P \to \lceil P \rceil$.

Proof.

$$\begin{aligned} &\vdash P \to \lceil P \rceil \\ \text{iff } &\vdash \forall x.(x \in P \to \lceil P \rceil) \\ \text{if } &\vdash x \in P \to \lceil P \rceil \\ \text{iff } &\vdash x \in P \to x \in \lceil P \rceil \\ \text{iff } &\vdash x \in P \to \exists y.(y \in P \wedge x \in \lceil y \rceil) \\ \text{iff } &\vdash x \in P \to \neg\forall y.(y \notin P \vee x \notin \lceil y \rceil) \\ \text{iff } &\vdash \forall y.(y \notin P \vee x \notin \lceil y \rceil) \to x \notin P \\ \text{if } &\vdash x \notin P \vee x \notin \lceil x \rceil \to x \notin P \\ \text{iff } &\vdash x \in P \to x \in P \wedge x \in \lceil x \rceil \\ \text{iff } &\vdash x \in P \to x \in \lceil x \rceil \\ \text{if } &\vdash x \in \lceil x \rceil \end{aligned}$$

Remark Similarly we can show $\vdash \lfloor P \rfloor \to P$. □

Proposition 21. $\vdash \forall x.(x \in P) = \lfloor P \rfloor$, *where x occurs free in P.*

Proof. By Proposition 13 and 11, it suffices to show

$$\vdash \forall x.(x \in P) \to \lfloor P \rfloor \tag{2}$$

and

$$\vdash \lfloor P \rfloor \to \forall x.(x \in P). \tag{3}$$

To show (2),

$$
\begin{aligned}
&\vdash \forall x.(x \in P) \to \lfloor P \rfloor \\
\text{iff } &\vdash \forall x.\lceil x \wedge P \rceil \to \neg\lceil \neg P \rceil \\
\text{iff } &\vdash \lceil \neg P \rceil \to \exists x.\neg\lceil x \wedge P \rceil \\
\text{iff } &\vdash \forall y.(y \in (\lceil \neg P \rceil \to \exists x.\neg\lceil x \wedge P \rceil)) \\
\text{if } &\vdash y \in (\lceil \neg P \rceil \to \exists x.\neg\lceil x \wedge P \rceil \\
\text{iff } &\vdash \exists z_1.(z_1 \notin P \wedge y \in \lceil z_1 \rceil) \to \\
&\qquad \exists x.\neg(\exists z_2.(z_2 = x \wedge z_2 \in P \wedge y \in \lceil z_2 \rceil)) \\
\text{iff } &\vdash \exists z_1.(z_1 \notin P \wedge \top) \to \qquad \text{(Proposition 19, \textbf{??}, and Corollary \textbf{??})} \\
&\qquad \exists x.\neg(\exists z_2.(z_2 = x \wedge z_2 \in P \wedge \top)) \\
\text{iff } &\vdash \exists z_1.(z_1 \notin P) \to \exists x.\neg(\exists z_2.(z_2 = x \wedge z_2 \in P)) \\
\text{iff } &\vdash \forall x.(\exists z_2.(z_2 = x \wedge z_2 \in P)) \to \forall z_1.(z_1 \in P) \\
\text{if } &\vdash \forall z_1.(\forall x.(\exists z_2.(z_2 = x \wedge z_2 \in P)) \to (z_1 \in P)) \\
\text{if } &\vdash \forall x.(\exists z_2.(z_2 = x \wedge z_2 \in P)) \to (z_1 \in P).
\end{aligned}
$$

Since $\vdash \forall x.(\exists z_2.(z_2 = x \wedge z_2 \in P)) \to \exists z_2.(z_2 = z_1 \wedge z_2 \in P)$, it suffices to show

$$
\begin{aligned}
&\vdash \exists z_2.(z_2 = z_1 \wedge z_2 \in P) \to (z_1 \in P) \\
\text{iff } &\vdash z_1 \notin P \to \forall z_2.(z_2 \neq z_1 \vee z_2 \notin P) \\
\text{if } &\vdash \forall z_2.(z_1 \notin P \to z_2 \neq z_1 \vee z_2 \notin P) \\
\text{if } &\vdash z_1 \notin P \to z_2 \neq z_1 \vee z_2 \notin P \\
\text{if } &\vdash z_2 = z_1 \wedge z_2 \in P \to z_1 \in P.
\end{aligned}
$$

And we proved (2).

Similarly, to show (3),

$$
\begin{aligned}
&\vdash \lfloor P \rfloor \to \forall x.(x \in P) \\
\text{iff } &\vdash \exists x.\neg\lceil x \wedge P \rceil \to \lceil \neg P \rceil \\
\text{iff } &\vdash \forall y.(y \in \exists x.\neg\lceil x \wedge P \rceil \to \lceil \neg P \rceil) \\
\text{if } &\vdash y \in \exists x.\neg\lceil x \wedge P \rceil \to \lceil \neg P \rceil \\
\text{iff } &\vdash \exists x.\neg\exists z_2.(z_2 = x \wedge z_2 \in P) \to \exists z_1.(z_1 \notin P) \\
\text{iff } &\vdash \forall z_1.(z_1 \in P) \to \exists z_2.(z_2 = x \wedge z_2 \in P) \\
\text{iff } &\vdash x \in P \to \exists z_2.(z_2 = x \wedge z_2 \in P).
\end{aligned}
$$

We proved (3).

Remark If x occurs free in P, the result does not hold. For example, let P be $upto(x)$ where $upto(\cdot)$ is interpreted to $upto(n) = \{0, 1, \ldots, n\}$ on $\mathbb{N}$. □

Remark From Membership Introduction and Elimination inference rules and Proposition 21, $\vdash P$ iff $\vdash \lfloor P \rfloor$.

Proposition 22 (Classification Reasoning)**.** *For any P and Q, from $\vdash P \to Q$ and $\vdash \neg P \to Q$ deduce $\vdash Q$.*

Proof. From $\vdash \neg P \to Q$ deduce $\vdash \neg Q \to P$. Notice that $\vdash P \to Q$, so we have $\vdash \neg Q \to Q$, i.e., $\vdash \neg\neg Q \vee Q$ which concludes the proof. □

Corollary 23. *For any P_1, P_2, and Q are patterns with $\vdash P_1 \vee P_2$, from $\vdash P_1 \to Q$ and $\vdash P_2 \to Q$, deduce $\vdash Q$.*

Definition 24 (Predicate Pattern)**.** *A pattern P is called a predicate pattern or a predicate if $\vdash (P = \top) \vee (P = \bot)$.*

Remark Predicate patterns are closed under all logic connectives.

Remark For any P, $\lceil P \rceil$ is a predicate pattern.

Proposition 25. $\vdash (\lceil P \rceil = \bot) = (P = \bot)$ *and* $\vdash (\lfloor P \rfloor = \top) = (P = \top)$.

Proof. It is easy to prove one derivation from the other, so we only prove the first one. By Proposition 13, it suffices to prove

$$\vdash (\lceil P \rceil = \bot) \to (P = \bot) \tag{4}$$

and

$$\vdash (P = \bot) \to (\lceil P \rceil = \bot) \tag{5}$$

The proof of (5) is trivial and we left it as an exercise. We now prove (4) through the following backward reasoning.

$$\begin{aligned}
& \vdash (\lceil P \rceil = \bot) \to (P = \bot) \\
\text{iff} \quad & \vdash \forall y.(y \in ((\lceil P \rceil = \bot) \to (P = \bot))) \\
\text{if} \quad & \vdash y \in ((\lceil P \rceil = \bot) \to (P = \bot)) \\
\text{iff} \quad & \vdash (y \in (\lceil P \rceil = \bot) \to (y \in (P = \bot)).
\end{aligned} \tag{6}$$

While for any pattern Q,

$$\begin{aligned}
& \vdash y \in (Q = \bot) \\
\text{iff} \quad & \vdash y \in \neg\lceil \neg(Q \leftrightarrow \bot)\rceil \\
\text{iff} \quad & \vdash y \in \neg\lceil Q \rceil \\
\text{iff} \quad & \vdash \neg\exists z.(z \in Q \wedge y \in \lceil z \rceil) \\
\text{iff} \quad & \vdash \neg\exists z.(z \in Q)
\end{aligned}$$

So we continue to prove (6) by showing

$$\begin{array}{ll} & \vdash (y \in (\lceil P \rceil = \bot)) \to (y \in (P = \bot)) \\ \text{iff} & \vdash \neg\exists z.(z \in \lceil P \rceil) \to \neg\exists z.(z \in P) \\ \text{iff} & \vdash \exists z.(z \in P) \to \exists z.(z \in \lceil P \rceil) \\ \text{iff} & \vdash \exists z.(z \in P) \to \exists z.(\exists z_1.(z_1 \in P \wedge z \in \lceil z_1 \rceil)) \\ \text{iff} & \vdash \exists z.(z \in P) \to \exists z.\exists z_1.(z_1 \in P) \\ \text{iff} & \vdash \exists z_1.(z_1 \in P) \to \exists z.\exists z_1.(z_1 \in P). \end{array}$$

And we finish the proof by noticing the fact that for any pattern Q and variable x,

$$\vdash Q \to \exists x.Q.$$

□

Proposition 26. *For any predicate P, $\vdash (P \neq \top) = (P = \bot)$ and $\vdash (P \neq \bot) = (P = \top)$.*

Proof. We only prove the first derivation, by showing both

$$\vdash (P \neq \top) \to (P = \bot) \tag{7}$$

and

$$\vdash (P = \bot) \to (P \neq \top). \tag{8}$$

Proving (8) is trivial. We now prove (7), which is also trivial by transforming disjunction to implication. □

Proposition 27. *For any pattern Q and any predicate pattern P, $\vdash P \vee Q$ iff $\vdash P \vee \lfloor Q \rfloor$.*

Proof. ($\Leftarrow$) is obtained immediately by the remark of Proposition 20. We now prove ($\Rightarrow$).

Because $\vdash Q = \top \vee Q \neq \top$, it suffices to show

$$\vdash Q = \top \to (P \vee \lfloor Q \rfloor = \top) \tag{9}$$

and

$$\vdash Q \neq \top \to (P \vee \lfloor Q \rfloor = \top) \tag{10}$$

by Corollary 23, and the fact that $\vdash P \vee \lfloor Q \rfloor = \top$ and $\vdash \top$ imply $\vdash P \vee \lfloor Q \rfloor$.

The proof of (9) is straightforward as follows.

$$\begin{array}{ll} & \vdash Q = \top \to (P \vee \lfloor Q \rfloor = \top) \\ \text{if} & \vdash Q = \top \to (P \vee \lfloor \top \rfloor = \top) \\ \text{if} & \vdash Q = \top \to (\top = \top) \\ \text{if} & \vdash \top. \end{array}$$

The proof of (10) needs more effort:

$$\begin{aligned}
& \vdash Q \neq \top \to (P \vee \lfloor Q \rfloor = \top) \\
\text{iff} \quad & \vdash (Q = \top) \vee (P \vee \lfloor Q \rfloor = \top) \\
\text{iff} \quad & \vdash (\lfloor Q \rfloor = \top) \vee (P \vee \lfloor Q \rfloor = \top) \\
\text{iff} \quad & \vdash \lfloor Q \rfloor \neq \top \to (P \vee \lfloor Q \rfloor = \top) \\
\text{iff} \quad & \vdash \lfloor Q \rfloor = \bot \to (P \vee \lfloor Q \rfloor = \top) \\
\text{if} \quad & \vdash \lfloor Q \rfloor = \bot \to (P \vee \bot = \top) \\
\text{iff} \quad & \vdash \lfloor Q \rfloor = \bot \to (P = \top) \\
\text{if} \quad & \vdash Q = \top \vee P = \top.
\end{aligned}$$

Notice that P is a predicate pattern, so it suffices to show

$$\vdash P = \top \to (Q = \top \vee P = \top),$$

whose validity is obvious, and

$$\vdash P = \bot \to (Q = \top \vee P = \top),$$

which is proved by showing

$$\vdash P = \bot \to Q = \top. \tag{11}$$

Because $\vdash P \vee Q$, it suffices to show

$$\begin{aligned}
& \vdash P = \bot \to (P \vee Q) \to (Q = \top) \\
\text{if} \quad & \vdash P = \bot \to (\bot \vee Q) \to (Q = \top) \\
\text{iff} \quad & \vdash P = \bot \to Q \to (Q = \top) \\
\text{if} \quad & \vdash Q \to (Q = \top) \\
\text{iff} \quad & \vdash (Q \neq \top) \to \neg Q \\
\text{iff} \quad & \vdash (\lfloor Q \rfloor = \bot) \to \neg Q.
\end{aligned}$$

Notice we have $\vdash Q \to \lfloor Q \rfloor$, which means $\vdash \neg\lfloor Q \rfloor \to \neg Q$, so it suffices to show

$$\begin{aligned}
& \vdash (\lfloor Q \rfloor = \bot) \to \neg\lfloor Q \rfloor \\
\text{iff} \quad & \vdash (\lfloor Q \rfloor = \bot) \to \neg\bot \\
\text{iff} \quad & \vdash (\lfloor Q \rfloor = \bot) \to \top \\
\text{iff} \quad & \vdash \top.
\end{aligned}$$

And this concludes the proof. □

Theorem 28 (Deduction Theorem)**.** *If* $\Gamma \cup \{P\} \vdash Q$ *and the derivation does not use* $\forall x$*-Generalization where* x *is free in* P*, then* $\Gamma \vdash \lfloor P \rfloor \to Q$*.*

Proof. The proof is by induction on n, the length of the derivation of Q from $\Gamma \cup \{P\}$.

Base step: $n = 1$, and Q is an axiom, or P, or a member of Γ. If Q is an axiom or a member of Γ, then $\Gamma \vdash Q$ and as a result, $\Gamma \vdash \lfloor P \rfloor \rightarrow Q$. If Q is P, then $\Gamma \vdash \lfloor P \rfloor \rightarrow Q$ by Proposition 20.

Induction step: Let $n > 1$. Suppose that if P' can be deduced from $\Gamma \cup \{P\}$ without using $\forall x$-Generalization where x is free in P, in a derivation containing fewer than n steps, then $\Gamma \vdash \lfloor P \rfloor \rightarrow P'$.

Case 1: Q is an axiom, or P, or a member of Γ. Precisely as in the Base step, we show that $\vdash \lfloor P \rfloor \rightarrow Q$.

Case 2: Q follows from two previous patterns in the derivation by an application of Modus Ponens. These two patterns must have the forms Q_1 and $Q_1 \rightarrow Q$, and each one can certainly be deduced from $\Gamma \cup \{P\}$ by a derivation with fewer than n steps, by just omitting the subsequent members from the original derivation from $\Gamma \cup \{P\} \vdash Q$. So we have $\Gamma \cup \{P\} \vdash Q_1$ and $\Gamma \cup \{P\} \vdash Q_1 \rightarrow Q$, and, applying the hypothesis of induction, $\Gamma \vdash \lfloor P \rfloor \rightarrow Q_1$ and $\Gamma \vdash \lfloor P \rfloor \rightarrow (Q_1 \rightarrow Q)$. It follows immediately that $\Gamma \vdash \lfloor P \rfloor \rightarrow Q$.

Case 3: Q follows from a previous pattern in the derivation by an application of $\forall x_i$-Generalization where x_i does not occur free in P. So Q is $\forall x_i.Q_1$, say, and Q_1 appears previously in the derivation. Thus $\Gamma \cup \{P\} \vdash Q_1$, and the derivation has fewer than n steps, so $\Gamma \vdash \lfloor P \rfloor \rightarrow Q_1$, since there is no application of Universal Generalization involving a free variable of P. Also x_i cannot occur free in P, as it is involved in an application of Universal Generalization in the deduction of Q from $\Gamma \cup \{P\}$. So we have a derivation of $\Gamma \vdash \lfloor P \rfloor \rightarrow Q$ as follows.

$$\begin{aligned} & \Gamma \vdash \lfloor P \rfloor \rightarrow Q \\ \text{iff} \quad & \Gamma \vdash \lfloor P \rfloor \rightarrow \forall x_i.Q_1 \\ \text{if} \quad & \Gamma \vdash \forall x_i.(\lfloor P \rfloor \rightarrow Q_1) \\ \text{if} \quad & \Gamma \vdash \lfloor P \rfloor \rightarrow Q_1. \end{aligned}$$

So $\Gamma \vdash \lfloor P \rfloor \rightarrow Q$ as required.

Case 4: Q follows from a previous pattern in the derivation by an application of Membership Introduction. So Q is $\forall x_i.(x_i \in Q_1)$ with x_i is free in Q_1, say, and Q_1 appears previously in the derivation. Thus $\Gamma \cup \{P\} \vdash Q_1$, and the derivation has fewer than n steps, so $\Gamma \vdash \lfloor P \rfloor \rightarrow Q_1$, since there is no application of Universal Generalization involving a free variable of P. So we have a derivation of $\Gamma \vdash \lfloor P \rfloor \rightarrow Q$ as follows.

$$\begin{aligned} & \Gamma \vdash \lfloor P \rfloor \rightarrow Q \\ \text{iff} \quad & \Gamma \vdash \lfloor P \rfloor \rightarrow \forall x_i.(x_i \in Q_1) \\ \text{iff} \quad & \Gamma \vdash \lfloor P \rfloor \rightarrow \lfloor Q_1 \rfloor, \end{aligned}$$

which follows by the hypothesis of induction $\Gamma \vdash \lfloor P \rfloor \rightarrow Q_1$ and the fact that $\Gamma \vdash Q_1 \rightarrow \lfloor Q_1 \rfloor$ (by the Remark in Proposition 20).

Case 5: Q follows from a previous pattern in the derivation by an application of Membership Elimination. The previous pattern must have the form $\forall x_i.(x_i \in Q)$, and can be deduced from $\Gamma \cup \{P\}$ by a derivation with fewer than n steps, by just omitting

the subsequent members from the original derivation from $\Gamma \cup \{P\} \vdash Q$. So we have $\Gamma \cup \{P\} \vdash \forall x_i.(x_i \in Q)$, and, applying the hypothesis of induction, $\Gamma \vdash \lfloor P \rfloor \to \forall x_i.(x_i \in Q)$. So we have a derivation of $\Gamma \vdash \lfloor P \rfloor \to Q$ as follows.

$$\begin{array}{lll} & \Gamma \vdash \lfloor P \rfloor \to Q & \\ \text{iff} & \Gamma \vdash \neg\lfloor P \rfloor \vee Q & \\ \text{iff} & \Gamma \vdash \neg\lfloor P \rfloor \vee \lfloor Q \rfloor & \text{(Proposition 27)} \\ \text{iff} & \Gamma \vdash \neg\lfloor P \rfloor \vee \forall x_i.(x_i \in Q) & \\ \text{iff} & \Gamma \vdash \lfloor P \rfloor \to \forall x_i.(x_i \in Q), & \end{array}$$

which is the hypothesis of induction. And this concludes our inductive proof. □

Corollary 29 (Closed-form Deduction Theorem)**.** *If P is closed, $\Gamma \cup \{P\} \vdash Q$ implies $\Gamma \vdash \lfloor P \rfloor \to Q$.*

Theorem 30 (Frame Rule)**.** *Let $\sigma \in \Sigma$ be a symbol in the signature. From $P_1 \to P_2$, deduce $\sigma(P_1) \to \sigma(P_2)$. In its most general form, $P_1 \to P_2$ deduces $\sigma(Q_1, \ldots, P_1, \ldots, Q_n) \to \sigma(Q_1, \ldots, P_2, \ldots, Q_n)$.*

Proof. we write $\sigma(Q_1, \ldots, P_i, \ldots, Q_n)$ as $\sigma(P_i, \vec{Q})$ for short, for any $i \in \{1, 2\}$.

$$\begin{array}{lll} & \vdash \sigma(P_1, \vec{Q}) \to \sigma(P_2, \vec{Q}) & \\ \text{iff} & \vdash y \in (\sigma(P_1, \vec{Q}) \to \sigma(P_2, \vec{Q})) & \\ \text{iff} & \vdash (y \in \sigma(P_1, \vec{Q})) \to (y \in \sigma(P_2, \vec{Q})) & \\ \text{iff} & \vdash \exists z_1.\exists \vec{z}.(z_1 \in P_1 \wedge \vec{z} \in \vec{Q} \wedge y \in \sigma(z_1, \vec{z})) & \\ & \to \exists z_2.\exists \vec{z}.(z_2 \in P_2 \wedge \vec{z} \in \vec{Q} \wedge y \in \sigma(z_2, \vec{z})) & \\ \text{iff} & \vdash \exists z_1.\exists \vec{z}.(z_1 \in P_1 \wedge \vec{z} \in \vec{Q} \wedge y \in \sigma(z_1, \vec{z}) & \\ & \qquad \to z_1 \in P_2 \wedge \vec{z} \in \vec{Q} \wedge y \in \sigma(z_1, \vec{z})) & \\ \text{iff} & \vdash \exists z_1.\exists \vec{z}.(z_1 \in P_1 \to z_1 \in P_2) & \\ \text{if} & \vdash \exists z_1.(z_1 \in P_1 \to z_1 \in P_2) & \\ \text{if} & \vdash P_1 \to P_2. & \end{array}$$

□

Corollary 31 (Frame Rule)**.** $\vdash \lfloor P \to Q \rfloor \to (\sigma(P) \to \sigma(Q))$

2.3 The ML2FOL translation

Given a matching logic signature $(S, \Sigma_{func} \cup \Sigma_{partial} \cup \Sigma_{uninterpreted})$. Define a first-order logic signature (S, Φ, Π) whose has the same set of sorts S. The set of function symbols Φ and the set of predicate symbols Π are defined as follows.

- For any n-arity $\sigma \in \Sigma_{uninterpreted}$ whose argument sorts are $s_1, \ldots, s_n$ and result sort is s, introduce π_σ that is an $(n+1)$-arity predicate symbol in Π whose argument sorts are $s_1, \ldots, s_n, s$. Intuitively, $\pi_\sigma(x_1, \ldots, x_n, y)$ holds in FOL if $y \in \sigma(x_1, \ldots, x_n)$ holds in ML.

- For any n-arity $\sigma \in \Sigma_{func}$ whose argument sorts are $s_1, \ldots, s_n$ and result sort is s, introduce σ that is an n-arity function symbol in Φ whose has the same signature as $\sigma \in \Sigma_{func}$. Intuitively, the term $\sigma(x_1, \ldots, x_n)$ in FOL is the only element that is in the singleton $\sigma(x_1, \ldots, x_n)$ in ML.
- For any n-arity $\sigma \in \Sigma_{partial}$ whose argument sorts are $s_1, \ldots, s_n$ and result sort is s, introduce δ_σ that is an n-arity predicate symbol in Π and $\tilde{\sigma}$ that is an n-arity function symbol in Φ. Both δ_σ and $\tilde{\sigma}$ have the argument sorts $s_1, \ldots, s_n$, and $\tilde{\sigma}$ has the result sort s. Intuitively, $\delta_\sigma(x_1, \ldots, x_n)$ holds in FOL if $\sigma(x_1, \ldots, x_n)$ is not the empty set in ML. If $\delta_\sigma(x_1, \ldots, x_n)$ holds in FOL, then the term $\tilde{\sigma}(x_1, \ldots, x_n)$ is the only element that is in the singleton $\sigma(x_1, \ldots, x_n)$ in ML.

Define two formula transformations fol and fol_2 as follows.

Input: a matching logic pattern P
Output: a first order formula
generate a fresh variable r whose sort the same as P ;
return $\forall r.\mathsf{fol}_2(P, r)$;

Algorithm 1: The fol transformation

Input: a matching logic pattern P and a variable r
Output: a first order formula
switch *the pattern* P **do**
  case x **do**
    return $r = x$;
  case $Q_1 \wedge Q_2$ **do**
    return $\mathsf{fol}_2(Q_1, r) \wedge \mathsf{fol}_2(Q_2, r)$;
  case $\neg Q$ **do**
    return $\neg\mathsf{fol}_2(Q, r)$;
  case $\exists x.Q$ **do**
    return $\exists x.\mathsf{fol}_2(Q, r)$;
  case $\sigma(Q_1, \ldots, Q_n)$ *where* σ *is uninterpreted* **do**
    return $\exists r_1 \ldots r_n\, (\pi_\sigma(r1, \ldots, r_n, r) \wedge \mathsf{fol}_2(Q_1, r_1) \wedge \cdots \wedge \mathsf{fol}_2(Q_n, r_n))$;
  case $\sigma(Q_1, \ldots, Q_n)$ *where* σ *is functional* **do**
    return $\exists r_1 \ldots r_n\, (r = \sigma(r_1, \ldots, r_n) \wedge \mathsf{fol}_2(Q_1, r_1) \wedge \cdots \wedge \mathsf{fol}_2(Q_n, r_n))$;
  case $\sigma(Q_1, \ldots, Q_n)$ *where* σ *is partial* **do**
    return
      $\exists r_1 \ldots r_n\, (\delta_\sigma(r_1, \ldots, r_n) \wedge r = \tilde{\sigma}(r_1, \ldots, r_n) \wedge \mathsf{fol}_2(Q_1, r_1) \wedge \cdots \wedge \mathsf{fol}_2(Q_n, r_n))$
      ;
end

Algorithm 2: The fol_2 transformation

Proposition 32. *For any P, Q are patterns and r, r' are fresh variables, the following holds.*

- $\mathsf{fol}_2(P = Q, r) = \forall r'.(\mathsf{fol}_2(P, r') \leftrightarrow \mathsf{fol}_2(Q, r'))$.
- $\mathsf{fol}_2(\lceil P \rceil, r) = \exists r'.\mathsf{fol}_2(P, r')$.
- $\mathsf{fol}_2(\lfloor P \rfloor, r) = \forall r'.\mathsf{fol}_2(P, r')$.
- $\mathsf{fol}_2(P \subseteq Q, r) = \forall r'.(\mathsf{fol}_2(P, r') \rightarrow \mathsf{fol}_2(Q, r'))$.

Proof. To be continued. □

Theorem 33. *For any pattern set* Γ, Γ *is satisfiable in matching logic iff* $\mathsf{fol}(\Gamma)$ *is satisfiable in first-order logic.*

In the following, we introduce a matching logic theory of heaps as a running example to show how ML2FOL works. From now on, we will use s-expressions to write matching logic patterns and first-order formulas. Readers who are familiar with SMT-LIB should find our representation almost identical to the one that many SMT solvers, such as Z3, use. Lisp programmers should find our representation quite easy to understand, too.

```
; A matching logic theory of maps

(declare-sort Nat)
(declare-sort NatSeq)
(declare-sort Map)

; Natural numbers

(declare-func zero () Nat)
(declare-func succ (Nat) Nat)

(declare-func one        () Nat)
(declare-func two        () Nat)
...

(assert (= one        (succ zero     )))
(assert (= two        (succ one      )))
...

; succ is injective
(assert (forall ((x Nat) (y Nat))
  (= (= (succ x) (succ y))
     (= x y))))

; succ(x) =/= x
(assert (forall ((x Nat))
  (not (= (succ x) x))))

; Sequence of naturals

(declare-func epsilon () NatSeq)
(declare-func cons (Nat NatSeq) NatSeq)
(declare-func append (NatSeq NatSeq) NatSeq)

(assert (forall ((x Nat) (s NatSeq))
  (not (= (cons x s) s))))
```

```
(assert (forall ((x1 Nat) (x2 Nat) (s1 NatSeq) (s2 NatSeq))
  (= (= (cons x1 s1) (cons x2 s2))
     (and (= x1 x2) (= s1 s2)))))

(assert (forall ((s1 NatSeq) (s2 NatSeq) (s3 NatSeq))
  (= (append (append s1 s2) s3)
     (append s1 (append s2 s3)))))

(assert (forall ((s NatSeq))
  (= (append s epsilon) s)))

(assert (forall ((s NatSeq))
  (= (append epsilon s) s)))

(assert (forall ((s1 NatSeq) (s2 NatSeq) (x Nat))
  (= (append (cons x s1) s2)
     (cons x (append s1 s2)))))

(declare-func rev (NatSeq) NatSeq)

(assert (= (rev epsilon) epsilon))

(assert (forall ((x Nat) (s NatSeq))
  (= (rev (cons x s))
     (append (rev s) (cons x epsilon)))))

; Maps

(declare-func emp () Map)

; x |-> y
(declare-part mapsto (Nat Nat) Map)

; 0 |-> y is bottom
(assert (forall ((y Nat))
  (not (mapsto zero y))))

; succ(x) |-> y is defined
(assert (forall ((x Nat) (y Nat))
  (ceil (mapsto (succ x) y))))

; succ(x1) |-> y1 = succ(x2) |-> y2 iff x1 = x2 /\ y1 = y1
(assert (forall ((x1 Nat) (x2 Nat) (y1 Nat) (y2 Nat))
  (= (= (mapsto (succ x1) y1) (mapsto (succ x2) y2))
     (and (= x1 x2) (= y1 y2)))))

; merge is a partial AC binary function
(declare-part merge (Map Map) Map)

; commutativity
(assert (forall ((h1 Map) (h2 Map))
(= (merge h1 h2) (merge h2 h1))))

; associativity
(assert (forall ((h1 Map) (h2 Map) (h3 Map))
  (= (merge (merge h1 h2) h3)
```

```
     (merge h1 (merge h2 h3)))))
; identity
(assert (forall ((h Map))
  (= h (merge h emp))))

; x |-> y * x |-> z = bottom
(assert (forall ((x Nat) (y Nat) (z Nat))
  (not (merge (mapsto x y) (mapsto x z)))))

; mapstoseq
(declare-part mapstoseq (Nat NatSeq) Map)

(assert (forall ((x Nat))
  (= (mapstoseq x epsilon) emp)))

(assert (forall ((x Nat) (y Nat) (s NatSeq))
  (= (mapstoseq x (cons y s))
     (merge (mapsto x y) (mapstoseq (succ x) s)))))

(declare-symb list (Nat NatSeq) Map)

(assert (forall ((x Nat))
  (= (list x epsilon)
     (and emp (= x zero)))))

(assert (forall ((x Nat) (y Nat) (s NatSeq))
  (= (list x (cons y s))
     (exists ((z Nat))
       (merge (mapstoseq x (cons y (cons z epsilon)))
              (list z s))))))
```

Validity and satisfiability

The duality between validity and satisfiability forms the foundation of SMT solvers. Suppose one wants to deduce φ from a set of axioms Γ. What he or she can do is adding the negation of the proof obligation $\neg\varphi$ to the axiom set and try to prove $\Gamma \cup \{\neg\varphi\}$ is unsatisfiable. This approach is justified thanks to the next theorem that establishes a duality between satisfiability and validity.

Theorem 34. *For any first-order theory* Γ *and first-order formula* φ*,*

$$\Gamma \models \varphi \quad \textit{iff} \quad \Gamma \cup \{\neg\varphi\} \textit{ is unsatisfiable.}$$

The dual relation between validity and satisfiability is less straightforward then the one in first-order logic.

Theorem 35. *For any matching logic theory* Γ *and a closed pattern* P*,*

$$\Gamma \models \varphi \quad \textit{iff} \quad \Gamma \cup \{\neg\lfloor\varphi\rfloor\} \textit{ is unsatisfiable.}$$

Proof. We here give a proof using the deduction theorem (Theorem 28). □

Simplification

The formula(s) that are autogenerated by ML2FOL are often unnecessarily verbose. Quantified dummy variables are everywhere, which makes it too hard for SMT solvers. Therefore, it is necessary to preprocess and simplify the autogenerated formulas before throwing them to solvers. In the next paragraph, we will introduce two main simplification rules that are crucial and useful in practice. All of them are trivially valid simplification rules, so the emphasis will be put on the reason why we apply them.

Rule 1: Eliminating existential quantifiers For any term t where x does not occur,

$$\exists x.((x = t) \wedge \varphi) \equiv \varphi[x := t].$$

This rule is often used to simplify formulas $\mathsf{fol}_2(\sigma(P), r)$.

Rule 2: Eliminating universal quantifiers For terms t_1, t_2 where x does not occur,

$$\forall x.(x = t_1 \leftrightarrow x = t_2) \equiv t_1 = t_2.$$

This rule is often used to simplify formulas $\mathsf{fol}_2(P = Q, r)$ where P, Q are functional patterns.

Rule 2′ For any terms t_1, t_2, formulas φ_1, φ_2 where x does not occur (free), and formulas $\psi_1(x), \psi_2(x)$ which are satisfiable w.r.t. the variable x, [1]

$$\begin{aligned}&\forall x.\,((\varphi_1 \wedge \psi_1(x)) \leftrightarrow (\varphi_2 \wedge \psi_2(x)))\\ &\equiv (\varphi_1 \leftrightarrow \varphi_2) \wedge ((\varphi_1 \vee \varphi_2) \rightarrow \forall x.(\psi_1(x) \leftrightarrow \psi_2(x))).\end{aligned}$$

This rule is often used to simplify formulas $\mathsf{fol}_2(P = Q, r)$ where P, Q are partial patterns. Usually, $\psi_i(x)$ has the form $x = t_i$ with t_i is a term not containing x.

The main objective of the above simplification rules is to eliminate quantifiers of dummy variables that are autogenerated by the ML2FOL translation. In practice, we found cases where quantifier elimination is feasible but the above rules do not capture that. For example, when translating $\neg(P \subseteq Q)$ where P is a functional or partial pattern, one often gets

$$\neg\forall x.((\varphi \wedge x = t) \rightarrow \psi).$$

Although the formula cannot be simplified by any rules above, we can transform it to an equivalent formula that can be simplified, as follows.

$$\begin{aligned}&\neg\forall x.((\varphi \wedge x = t) \rightarrow \psi)\\ &\equiv \exists x.\neg((\varphi \wedge x = t) \rightarrow \psi)\\ &\equiv \exists x.(\varphi \wedge x = t \wedge \neg\psi)\\ &\equiv \varphi[x := t] \wedge \neg\psi[x := t].\end{aligned}$$

[1] That is, $\exists x.\psi_i(x)$ is valid, for $i = 1, 2$.

Performance

In general, the ML2FOL translation performs well on functional and partial patterns. This is because the translation takes special care of functional and partial symbols and encodes them in an efficient way in first-order logic. Translating arbitrary patterns is feasible but there is little benefit we can get from the translation. State-of-the-art SMT solvers tend to have a hard time dealing with the autogenerated first-order theory if the original matching logic theory has a heavy use of uninterpreted symbols that are neither functional nor partial. One typical example is the list theory in matching logic.

The author did some experiments with the translation and used the SMT solver Z3 to solve the autogenerated first-order theories. The experiment results are enclosed here, divided into three parts.

Part 1: functional symbols The author tested the performance of solving theories about functional patterns using the theory of sequence, defined below.

```
(declare-sort NatSeq)

(declare-func epsilon () NatSeq)
(declare-func cons (Nat NatSeq) NatSeq)
(declare-func append (NatSeq NatSeq) NatSeq)
(declare-func rev (NatSeq) NatSeq)

(assert (forall ((x Nat) (s NatSeq))
  (not (= (cons x s) s))))

(assert (forall ((x1 Nat) (x2 Nat) (s1 NatSeq) (s2 NatSeq))
  (= (= (cons x1 s1) (cons x2 s2))
     (and (= x1 x2) (= s1 s2)))))

(assert (forall ((s1 NatSeq) (s2 NatSeq) (s3 NatSeq))
  (= (append (append s1 s2) s3)
     (append s1 (append s2 s3)))))

(assert (forall ((s NatSeq))
  (= (append s epsilon) s)))

(assert (forall ((s NatSeq))
  (= (append epsilon s) s)))

(assert (forall ((s1 NatSeq) (s2 NatSeq) (x Nat))
  (= (append (cons x s1) s2)
     (cons x (append s1 s2)))))

(assert (= (rev epsilon) epsilon))

(assert (forall ((x Nat) (s NatSeq))
  (= (rev (cons x s))
     (append (rev s) (cons x epsilon)))))
```

It takes 1.00 second to prove rev([3;14;15;9;2;7;18;2;8;18;20;3;6;8;8]) = [8;8;6;3;20;18;8;2;18;7;2;9;15;14;3].

Part 2: partial symbols The author used the theory of map, where the separating conjunction operator merge is an associative and commutative partial symbol, to test

associative-commutative matching of partial symbols.

number of bindings	time in second	memory
8	12.79	574.42
9	91.35	2518.37

Part 3: uninterpreted symbols

example	time in second	memory
$list(13, [11])$	0.06	6.06
$list(5, [9; 2])$	1.49	33.45
$list(5, [9; 2; 5])$	212.77	172.01
$list(5, [9; 2; 5])$ with domain theory	0.67	52.73
$list(5, [13; 17; 9; 2; 5; 13])$ with domain theory	*n.a.*	*n.a.*

2.4 Propositional proof system

2.4.1 A complete proof system of propositional matching logic

Following previous proofs, we introduce the next complete proof system of propositional matching logic, which consists of five axiom schemata and one inference rule.

- (A1) $P \to (Q \to P)$
- (A2) $(P \to (Q \to R)) \to (P \to Q) \to (P \to R)$
- (A3) $(\neg Q \to \neg P) \to (P \to Q)$
- (K) $\sigma(\dots P \vee Q \dots) \leftrightarrow \sigma(\dots P \dots) \vee \sigma(\dots Q \dots)$
- (N) $\neg\sigma(\dots \bot \dots)$
- (MP) From P and $P \to Q$, deduce Q.
- (CG) From $P \leftrightarrow Q$, deduce $R[P] \leftrightarrow R[Q]$.

In this section, we consider the propositional fragment of matching logic, and its proof system. This work is closely related to polyadic model logic. The main technical methods are from there.

Before we introduce the proof system for propositional matching logic, let us define the syntax of propositional patterns as shown in the next grammar.

$$\begin{aligned} P ::= & P_1 \wedge P_2 \\ & \mid \neg P \\ & \mid \sigma(P_1, \dots, P_n). \end{aligned}$$

Note that the only *atomic* patterns in propositional matching logic are of the form $\sigma()$ where σ is a zero-arity symbol.

Our goal is to find a minimal proof system for propositional matching logic that is sound and complete. We will prove its completeness using the standard approach of *maximal consistent sets* from which we can build a *canonical model*. To help us focus on the real business, let us prove the completeness *before* introducing the proof system. The reader should think of the proof system as one that makes our proofs work. The sound and complete proof system are shown in Section 2.4.1 after we prove its completeness.

In the following, we use Γ to denote a set of patterns.

One key fact about our proof system is that the deduction theorem holds.

Theorem 36. *For any patterns P and Q that belong to the same sort, $\Gamma \cup \{Q\} \vdash P$ iff $\Gamma \vdash Q \to P$.*

Thus the reader can think of the proof system as the one that has only one inference rule: the modus ponens rule, together with the three axioms in the Hilbert propositional calculus.

Definition 37. *Γ is consistent if $\Gamma \nvdash \bot_s$ for any sort s.*

Definition 38. *Γ is maximal consistent if it is consistent and for any $\Gamma' \supsetneq \Gamma$, Γ' is inconsistent.*

Proposition 39. *If Γ is consistent and $\Gamma \nvdash P$, then $\Gamma \cup \{\neg P\}$ is consistent.*

Proof. Suppose $\Gamma \cup \{\neg P\}$ is inconsistent. By definition, $\Gamma \cup \{\neg P\} \vdash \bot$. By deduction theorem, $\Gamma \vdash \neg P \to \bot$, that is, $\Gamma \vdash P$, which contradicts the fact that $\Gamma \nvdash P$. □

Proposition 40. *If Γ is maximal consistent, then $P \in \Gamma$ iff $\Gamma \vdash P$.*

Proof. ($\Rightarrow$) is trivial. Let us prove ($\Leftarrow$). Since Γ is consistent and $\Gamma \vdash P$, we know $\Gamma \cup \{P\}$ is also consistent. Since Γ is maximal consistent, we know $\Gamma = \Gamma \cup \{P\}$, which implies that $P \in \Gamma$. □

Proposition 41. *If Γ is maximal consistent, then for any P, either $\Gamma \vdash P$ or $\Gamma \vdash \neg P$.*

Proof. Suppose $\Gamma \nvdash P$. Since Γ is consistent, we know $\Gamma \cup \{\neg P\}$ is also consistent. Since Γ is maximal consistent, we know $\Gamma = \Gamma \cup \{\neg P\}$, which implies that $\Gamma \vdash \neg P$. □

Proposition 42. *If $\Gamma \vdash P$, then $\Gamma \cup \{\neg P\}$ is inconsistent.*

Proposition 43. *For any Γ is consistent, there exists an extension $\Gamma' \supseteq \Gamma$ such that Γ' is maximal consistent. We call such Γ' the maximal consistent extension of Γ.*

Proof. Enumerate all patterns as $P_1, P_2, \ldots$. Define a sequence of sets $\Gamma_0, \Gamma_1, \Gamma_2, \ldots$ as follows.

$$\Gamma_0 = \Gamma,$$
$$\Gamma_i = \begin{cases} \Gamma_{i-1} \cup \{P_i\} & \text{if it is consistent} \\ \Gamma_{i-1} & \text{otherwise} \end{cases}.$$

Then $\Gamma = \Gamma_0 \subseteqq \Gamma_1 \subseteqq \ldots$ is a monotonic sequence of consistent sets. Let $\Gamma' = \bigcup \Gamma_i$. It is obvious that $\Gamma' \supseteqq \Gamma$. We are going to show that Γ' is maximal consistent.

Suppose Γ' is inconsistent. By definition, $\Gamma' \vdash \bot$. The derivation of $\bot$ must only use a finite number of patterns in Γ', which means there is a finite subset of Γ' that is inconsistent. Therefore, there must exist an $n > 0$ such that Γ_n is inconsistent, which contradicts the way we construct the sequence $\{\Gamma_i\}_i$.

Suppose Γ' is not maximal consistent. By definition, there exists a P such that $\Gamma' \nvdash P$ and $\Gamma' \cup \{P\}$ is also consistent. Rememeber we have enumerated all patterns as $P_1, P_2, \ldots$, so the pattern P must be one of them. Suppose $P = P_n$. Since $P = P_n \notin \Gamma'$, we know Γ_n does not include P_n. By the way of how we construct $\{\Gamma_i\}_i$, it must be the case that $\Gamma_{n-1} \cup \{P_n\}$ is inconsistent, which contradicts the fact that $\Gamma' \cup \{P\}$ is consistent. □

Definition 44. *Suppose Γ is maximal consistent. Its canonical model M_Γ is defined as follows.*

- *For each sort $s \in S$, the carrier set $M_s = \{\Gamma_s\}$ is a singleton set;*
- *For each symbol $\sigma \in \Sigma_{s_1,\ldots,s_n,s}$, its interpretation $\sigma_M \colon M_{s_1} \times \cdots \times M_{s_n} \to \rho(M_s)$ is defined as*

$$\sigma_M(\Gamma_{s_1}, \ldots, \Gamma_{s_n}) = \begin{cases} \{\Gamma_s\} & \text{if for any } P_1 \in \Gamma_{s_1}, \ldots, P_n \in \Gamma_{s_n},\ \sigma(P_1, \ldots, P_n) \in \Gamma_s \\ \emptyset & \text{otherwise} \end{cases}.$$

Proposition 45. *Suppose M is a canonical model. Then M is negation complete, in the sense that for any pattern P, either $M \vDash P$ or $M \vDash \neg P$.*

Proof. Simply notice that all the carrier sets of M are singleton sets. □

Proposition 46. *Suppose Γ is maximal consistent and M_Γ is its canonical model. For any pattern P, $M \vDash P$ iff $P \in \Gamma$.*

Proof. Let us conduct structural induction on P.

(Base case). P is atomic, that is, $P = \sigma()$ for some zero-arity symbol σ.

$$\begin{aligned} M \vDash \sigma() \quad &\text{iff} \quad \sigma_M() = \{\Gamma_s\} \\ &\text{iff} \quad \sigma() \in \Gamma_s. \end{aligned}$$

(Induction step). P is of the form $\neg Q$, $Q_1 \wedge Q_2$, or $\sigma(Q_1, \ldots, Q_n)$.

(Case A) $P = \neg Q$.

$$\begin{aligned} M \vDash \neg Q \quad &\text{iff} \quad \bar{\rho}(\neg Q) = \{\Gamma_s\} \\ &\text{iff} \quad \bar{\rho}(Q) = \emptyset \\ &\text{iff} \quad Q \notin \Gamma_s \\ &\text{iff} \quad \neg Q \in \Gamma_s. \end{aligned}$$

(Case B) $P = Q_1 \wedge Q_2$.

$$\begin{aligned} M \vDash Q_1 \wedge Q_2 \quad & \text{iff} \quad \bar{\rho}(Q_1 \wedge Q_2) = \{\Gamma_s\} \\ & \text{iff} \quad \bar{\rho}(Q_1) = \bar{\rho}(Q_2) = \{\Gamma_s\} \\ & \text{iff} \quad Q_1, Q_2 \in \Gamma_s \\ & \text{iff} \quad Q_1 \wedge Q_2 \in \Gamma_s. \end{aligned}$$

(Case C) $P = \sigma(Q_1, \ldots, Q_n)$.

$$\begin{aligned} M \vDash \sigma(Q_1, \ldots, Q_n) \quad & \text{iff} \quad \bar{\rho}(\sigma(Q_1, \ldots, Q_n)) = \{\Gamma_s\} \\ & \text{iff} \quad \text{for any } 1 \leq i \leq n,\ \bar{\rho}(Q_i) = \{\Gamma_{s_i}\} \\ & \qquad \text{and } \sigma_M(\Gamma_{s_1}, \ldots, \Gamma_{s_n}) = \{\Gamma_s\} \\ & \text{iff} \quad \text{for any } 1 \leq i \leq n,\ Q_i \in \Gamma_{s_i} \\ & \qquad \text{and for any } P_i \in \Gamma_{s_i},\ \sigma(P_1, \ldots, P_n) \in \Gamma_s \\ & \text{implies} \quad \sigma(Q_1, \ldots, Q_n) \in \Gamma_s. \end{aligned}$$

We need a little bit trick to finish our proof of (Case C). In the last step, we showed only one direction of implication. We need to prove the other direction, that is, $\sigma(Q_1, \ldots, Q_n) \in \Gamma_s$ implies that for any $1 \leq i \leq n$, $Q_i \in \Gamma_{s_i}$, and for any $P_i \in \Gamma_{s_i}$, $\sigma(P_1, \ldots, P_n) \in \Gamma_s$.

Let us first show the simpler one that every Q_i belongs to Γ_{s_i}. Assume the opposite. Without loss of generality, let us assume $Q_1 \notin \Gamma_{s_1}$. From Γ_{s_1}'s maximal consistency, we know that $\neg Q_1 \in \Gamma_{s_1}$, in other words, $\Gamma \vdash \neg Q_1$. The proof system (that we haven't seen) should let us deduce from that $\Gamma \vdash \neg\sigma(Q_1, \ldots, Q_n)$, which contradicts Γ's consistency.

We are going to prove the second point that $\sigma(P_1, \ldots, P_n) \in \Gamma_s$ for any $P_i \in \Gamma_{s_i}$. The proof needs a trick which I firstly learned from Natalia Gabriela Moanga's thesis *Many-sorted polyadic modal logic* [], where Moanga uses it to prove a similar result called the *Existence Lemma*.

We know that $\sigma(Q_1, \ldots, Q_n) \in \Gamma_s$ and that every Q_i belongs to Γ_{s_i}. We are going to consistently maximize $\{Q_i\}$ as we did in Proposition **??**, until we reach Γ_{s_i}. To do that, we enumerate all patterns as $P_1, P_2, \ldots$.

$$\begin{aligned} \Gamma \vdash \sigma(Q_1, \ldots, Q_n) \quad & \text{iff} \quad \Gamma \vdash \sigma(Q_1 \wedge (P_1 \vee \neg P_1), \ldots, Q_n \wedge (P_1 \vee \neg P_1)) \\ & \text{iff} \quad \Gamma \vdash \bigvee_{\sim_i \in \{+,-\}} \sigma(Q_1 \wedge {\sim_1} P_1, \ldots, Q_n \wedge {\sim_n} P_1) \\ & \text{iff} \quad \Gamma \vdash \sigma(Q_1 \wedge {\sim_1} P_1, \ldots, Q_n \wedge {\sim_n} P_1) \text{ for some } {\sim_i} \in \{+, -\}. \end{aligned}$$

Repeat the procedure for $P_2, P_3, \ldots$ until we obtain $\Pi^i = \{Q_i, [\neg]P_1, [\neg]P_2, \ldots\}$ for each $1 \leq i \leq n$. It is easy to show that Π^i is maximal consistent, by noticing that Π^i is negation complete and consistent.

In fact, $\Pi^i = \Gamma_{s_i}$. If not, then there exists a pattern P_i^* such that $P_i^* \in \Pi^i$ and $\neg P_i^* \in \Gamma_{s_i}$. By the contruction of Π^i, we can prove that $\Gamma \vdash \sigma(Q_1, \ldots, P_i^*, \ldots, Q_n)$, while from $\neg P_i^* \in \Gamma_{s_i}$ we can deduce $\Gamma \vdash \neg\sigma(Q_1, \ldots, P_i^*, \ldots, Q_n)$. This contradition shows that $\Pi^i = \Gamma_{s_i}$, which directly follows that $\Gamma \vdash \sigma(P_1, \ldots, P_n)$ for any $P_i \in \Gamma_{s_i} = \Pi^i$.

And here ends the whole proof.

□

Theorem 47. *For any set of patterns* Γ *and any pattern* P*, if* $\Gamma \vDash P$ *then* $\Gamma \vdash P$*.*

Proof. The case where Γ is inconsistent is trivial, so let us suppose Γ is consistent. Suppose there is a pattern P such that $\Gamma \vDash P$ but $\Gamma \nvdash P$. By Proposition **??**, $\Gamma \cup \{\neg P\}$ is consistent. Let Γ' be the maximal consistent extension of $\Gamma \cup \{\neg P\}$, and M be the canonical model of Γ'. Note that Γ' is consistent, so $\Gamma' \nvdash P$, and, by Proposition 46, $M \nvDash P$. On the other hand, $M \vDash \Gamma'$, which implies that $M \vDash \Gamma$ because $\Gamma \subseteq \Gamma'$. This contradicts against the fact that $\Gamma \vDash P$, because we found a model M which satisfies Γ but does not entail P. □

3 Applications

3.1 Axiomatizing transition systems

Definition 48. *A transition system* $T = (Cfg, \tau)$ *has a (finite or infinite) set Cfg of configurations and a (partial) transition function* $\tau : Cfg \rightharpoonup Cfg$*. The set of* direct successors *and* direct predecessors *of a configuration* $c \in Cfg$ *are denoted as* $\tau(c)$ *and* $\tau^{-1}(c)$ *respectively. A configuration is* terminal *if it has no successor.*

Remark 49. *There is a dual way to define a transition system* $T = (Cfg, \tau)$*. Instead of specifying the transition function* τ*, we can define the* reverse transition function *or the* predecessor function τ^{-1}*. This fact is witnessed by the next Galois connection*

$$\tau(c) = \{d \in Cfg \mid c \in \tau^{-1}(d)\},$$
$$\tau^{-1}(d) = \{c \in Cfg \mid d \in \tau(c)\}.$$

Fix a signature (S, Σ) where $S = \{Cfg\}$ has only one sort and $\Sigma = \{\bigcirc\}$ has a unary symbol. Such a signature is called *the signature of transition systems*, as illustrated by the obvious interpretation that takes the sort Cfg to the carrier set Cfg of a transition system (Cfg, τ) and interprets the symbol $\bigcirc$ as the predecessor function τ^{-1}.

3.1.1 One-path reachability

Definition 50. *Given* (S, Σ) *the a signature of transition systems. Introduce predicate patterns* $P \Rightarrow_1^{\exists} Q$*, read as "P one-step rewrites to Q", as syntactic sugar of* $P \to \bigcirc Q$*. A transition system theory is a theory whose signature is the signature of transition systems, and whose axioms are all of the form* $P \Rightarrow_1^{\exists} Q$*.*

The unary symbol $\bigcirc$ is known as the "strong-next" temporal connective in temporal logics. Its dual version, the so-called "weak-next" connective $\bullet$ is defined as $\neg\bigcirc\neg$ in the usual way.

Example 51. *Give* T *is a theory of transition systems, then* $T \vdash P \to \bullet\bot$ *implies that in any transition system* T *who is a model of the theory,* P *is a set of terminal configurations. For the same reason,* $T \vdash P \to \bigcirc\top$ *implies that* P *is a set of nonterminal configurations.*

Example 52. *Let T be a transition system whose configuration set contains all natural numbers plus one single special configuration c_{rand}. Natural numbers are all terminal configurations. Intuitively, c_{rand} is the special configuration that will randomly rewrite to some natural number in a step.*

Suppose one has that intuition of randomness in mind, and starts to write a specification for that intuition. The followings are possible rules that could be written (we abbreviate $\Rightarrow_1^\exists$ as $\Rightarrow$).

1. $c_{rand} \Rightarrow \top$. [2]
2. $\forall x.(c_{rand} \Rightarrow x)$.
3. $c_{rand} \subseteq \forall x.\bigcirc x$.

Before we study the subtle difference among these specifications, let us list some patterns that are logically equivalent to one of the above, which will cover most cases that any reader might think of. Patterns that are equivalent to the first case include $c_{rand} \Rightarrow \exists x.x$, $\exists x.(c_{rand} \Rightarrow x)$, and $c_{rand} \subseteq \exists x.\bigcirc x$. Patterns that are equivalent to the second are $c_{rand} \Rightarrow \forall x.x$, or $c_{rand} \Rightarrow \bot$. Patterns that are equivalent to the third include $\forall x.(c_{rand} \Rightarrow x)$.

Obviously, the second case makes no sense. In fact, it is a falsity, because c_{rand} will never be a subset of $\bigcirc\bot = \bot$, the empty set. The first case is simply saying that c_{rand} is nonterminating. This might seem counterintuitive, but it does capture some kind of randomness here: it allows any models *in which c_{rand} rewrites to something. Among those models, there exists the standard model where c_{rand} rewrites to possibly any of the natural numbers, but there also have models where c_{rand} only rewrite to some values but not others. It even allows models that rewrite c_{rand} to a fixed single value all the time. As a comparison, the third case precisely captures the standard model where c_{rand} is the predecessor of* all natural numbers.

To conclude, the third specification is the one that fits the best with the requirement of having some random objects that can potentially become any *possible objects, while the first specification is the one that allows* any *possible implementation of some nondeterministic objects. A typical example for that is the scheduler in a multithreaded program. The second specification is simply a mistake and should never be written.*

Traian: I'm not sure I understand why 2 and 3 are different. It seems to me that $c_{rand} \subseteq \forall x.\bigcirc x$ iff $c_{rand} \rightarrow \forall x.\bigcirc x$ iff $\forall x.c_{rand} \rightarrow \bigcirc x$ iff $\forall x.c_{rand} \Rightarrow x$

We can define $\Diamond P$, the temporal "eventually" operator with the semantics: if the computation terminates then a state where P holds is reachable. This can be written as:

$$\Diamond P = (\mu x.(P \vee \bigcirc x)) \vee (\nu x.\bigcirc x)$$

Definition 53. *It is natural to extend the one-step one-path reachability $P \Rightarrow_1^\exists Q$ relation to multisteps one-path reachability $P \Rightarrow^\exists Q$, defined as syntactic sugar for $P \rightarrow \Diamond Q$.*

[2] For the rewriting patterns below, we assume the right hand sides are patterns of the sort *Nat*, regarded as a subsort of *Cfg*, although matching logic is truly not an order-sorted logic. This is for the sake of the simplicity. Otherwise one may fall into unnecessary details that prevent him seeing the big picture.

3.1.2 All-path reachability

When dealing with nondeterministic programs, we are interested in a different kind of reachability. Instead of proving that there exists an execution path leading from P to Q we rather want that on all execution paths starting with P Q would eventually hold.

Note that $P \Rightarrow_1^\exists Q$ models the fact Q *can* be reached (i.e., there exists a path) from P in one step. To model all-path reachability, we first need to model that Q must hold if previously P held. We do that using the previously operator $\odot$ which is the inverse of $\bigcirc$ (thus representing the forward transition relation):

$$y \in \odot x \leftrightarrow x \in \bigcirc y.$$

Alternatively, $\odot$ can be defined as $\odot x = \exists y.y \wedge x \in \bigcirc y$ where y is distinct from x.

Using this we can express that if P previously held then Q must hold now.

Definition 54. *Let $P \Rightarrow_1^\forall Q$, read as "P must rewrite to Q", be the predicate pattern defined as the syntactic sugar for $\odot P \rightarrow Q$.*

As for the one-path relation, we want to extend this one-step relation to a relation $P \Rightarrow^\forall Q$. However, note that Q does not need to hold at the same rewrite depth for all paths. For example, consider the system $\{a \Rightarrow b, a \Rightarrow c, c \Rightarrow b\}$. We have that $a \Rightarrow^\forall b$, but there are two paths of different length leading a to b. Therefore we would like to define the all-path reachability relation as:

$$P \Rightarrow^\forall Q \iff (P \rightarrow Q) \vee (\odot(P \wedge \neg Q) \Rightarrow^\forall Q)$$

$$\_ \Rightarrow^\forall Q = \mu f.\lambda P.P \rightarrow Q \vee f(\odot(P \wedge \neg Q))$$

3.2 Symbolic execution

Different from normal concrete execution, symbolic execution method runs programs in a symbolic way, aiming for a better state-space converge rate. In the literatures of rewriting theory, symbolic execution is often called narrowing, whose main idea is to define a sequence of (symbolic) rewriting steps such that *any* concrete rewriting step can be seen as a specific way to *instantiate* the symbolic rewriting steps.

Definition 55. *A theory of symbolic execution is a theory of transition system whose axioms are all of the form $P(\vec{x}, \vec{z}) \Rightarrow_1^\forall Q(\vec{x}, \vec{y})$ where P, Q are term patterns with*

$$\vec{x} = \mathit{freevars}(P) \cap \mathit{freevars}(Q),$$
$$\vec{z} = \mathit{freevars}(P) \setminus \mathit{freevars}(Q),$$
$$\vec{y} = \mathit{freevars}(Q) \setminus \mathit{freevars}(P).$$

Axioms in a theory of symbolic execution are often called rules.

Definition 56. *Given T is a theory of symbolic execution with rules $P_i \Rightarrow_1^\forall Q_i$ for $1 \leq i \leq n$. Given R is a term pattern. The symbolic term that R rewrites to in one step is defined as*

$$\tau(R) = \bigvee_i \lceil R \wedge P_i \rceil \wedge Q_i.$$

Proposition 57. *The following propositions hold for term patterns and unconstrained symbols.*

- $\lceil P \wedge Q \rceil = (P = Q)$,
- $\lceil \sigma(P_1, \ldots, P_n) \wedge \sigma(Q_1, \ldots, Q_n) \rceil = \bigwedge_i (P_i = Q_i)$,

Proof. Notice that the second bullet is just a corollary of the first bullet and the fact that σ is unconstrained. □

Example 58. *Fix a signature of transition systems where the only sort is Nat, with the next two axioms*

$$2x \Rightarrow_1^{\forall} x,$$
$$2x + 1 \Rightarrow_1^{\forall} 6x + 4.$$

Start with a constant a. The symbolic term that a rewrites to in one step is

$$\begin{aligned}\tau(a) &= \lceil a \wedge 2x \rceil \wedge x \vee \lceil a \wedge 2x + 1 \rceil \wedge (6x + 4) \\ &= (a = 2x) \wedge x \vee (a = 2x + 1) \wedge (6x + 4).\end{aligned}$$

Example 59. *Given a signature of a simple programming language and the following rules.*

$$ite(true, s_1, s_2) \Rightarrow_1^{\forall} s_1,$$
$$ite(false, s_1, s_2) \Rightarrow_1^{\forall} s_2.$$

The symbolic term $ite(b, stmt_1, stmt_2)$ rewrites in one step to

$$\begin{aligned}\tau(ite(b, stmt_1, stmt_2)) &= \lceil ite(b, stmt_1, stmt_2) \wedge ite(true, s_1, s_2) \rceil \wedge s_1 \\ &\quad \vee \lceil ite(b, stmt_1, stmt_2) \wedge ite(false, s_1, s_2) \rceil \wedge s_2. \\ &= b = true \wedge stmt_1 = s_1 \wedge stmt_2 = s_2 \wedge s_1 \\ &\quad \vee b = false \wedge stmt_1 = s_1 \wedge stmt_2 = s_2 \wedge s_2 \\ &= b = true \wedge stmt_1 = s_1 \wedge stmt_2 = s_2 \wedge stmt_1 \\ &\quad \vee b = false \wedge stmt_1 = s_1 \wedge stmt_2 = s_2 \wedge stmt_2.\end{aligned}$$

3.3 Context

Fix a signature (S, Σ). For each sort $s \in S$, introduce an infinite number of *hole* variables, written as $\Box_1, \Box_2, \cdots$. Think of hole variables as normal matching logic variables, but lie in a disjoint namespace. Extend the grammar by adding the following.

$$\begin{aligned}P ::= &\cdots \\ &\mid \Box \\ &\mid \gamma\Box.P \\ &\mid P[P'].\end{aligned}$$

The sort of $\gamma\Box.P$ is the sort of P. The sort of $P[P']$ is the sort of P, too. Think of $\gamma\Box$ as a binder. Alpha-renaming is always assumed. Patterns of the form $\gamma\Box.P$ are often called *contexts*, denoted by $C, C_0, C_1, \ldots$. The pattern $P[P']$ is often called an *application*. The $\_[\_]$ operator is left associative.

Definition 60. *The context $\gamma\Box.\Box$ is called the identity context, denoted as I. Identity context has the axiom schema $\mathsf{I}[P] = P$ where P is any pattern.*

Example 61. $\mathsf{I}[\mathsf{I}] = \mathsf{I}$.

Example 62. *Consider a signature (S, Σ) of a simple imperative programming language, with $S = \{BExp, Pgm\}$, and $\Sigma = \{skip, ite, seq, true, false\}$. Add axiom schemata*

$$ite(C_1[B], P, Q) = (\gamma\Box.ite(C_1[\Box], P, Q))[B]$$

and

$$seq(C_2[P], Q) = (\gamma\Box.seq(C_2[\Box], Q))[P],$$

where P, Q are Pgm patterns, B is a $BExp$ pattern, C_1 is a $BExp$ context, and C_2 is a Pgm context.

Suppose we have the rewrite rules (schemata):

- $C[ite(true, P, Q)] \Rightarrow C[P]$,
- $C[ite(false, P, Q)] \Rightarrow C[Q]$,
- $C[seq(skip, Q)] \Rightarrow C[Q]$.

Example (a). Rewrite $seq(skip, skip)$.

$$\begin{aligned} seq(skip, skip) &= \mathsf{I}[seq(skip, skip)] \\ &\Rightarrow \mathsf{I}[skip] \\ &= skip. \end{aligned}$$

Example (b). Rewrite $ite(true, P, Q); R$.

$$\begin{aligned} seq(ite(true, P, Q), R) &= seq(\mathsf{I}[ite(true, P, Q)], R) \\ &= (\gamma\Box.seq(\mathsf{I}[\Box], R))[ite(true, P, Q)] \\ &\Rightarrow (\gamma\Box.seq(\mathsf{I}[\Box], R))[P] \\ &= seq(\mathsf{I}[P], R) \\ &= seq(P, R). \end{aligned}$$

Definition 63. *Let $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ is an n-arity symbol. We say σ is* active *on its ith argument ($1 \leq i \leq n$), if $\sigma(P_1, \ldots, C[P_i], \ldots, P_n) = (\gamma\Box.\sigma(P_1, \ldots, C[\Box], \ldots, P_n))[P_i]$. Orienting the equation from the left to the right is often called* heating*, while orienting from the right to the left is called* cooling.

Example 64. *Suppose f and g are binary symbols who are active on their first argument. Suppose a, b are constants, and x is a variable. Let $\Box_1$ and $\Box_2$ be two hole variables. Define two contexts $C_1 = \gamma\Box_1.f(\Box_1, a)$ and $C_2 = \gamma\Box_2.g(\Box_2, b)$.*

Because f is active on the first argument,

$$\begin{aligned} C_1[\varphi] &= (\gamma\Box_1.f(\Box_1, a))[\varphi] \\ &= (\gamma\Box_1.f(I[\Box_1], a))[\varphi] \\ &= f(I[\varphi], a) \\ &= f(\varphi, a), \text{ for any pattern } \varphi. \end{aligned}$$

And for the same reason, $C_2[\varphi] = g(\varphi, b)$. Then we have

$$\begin{aligned} C_1[C_2[x]] &= C_1[f(x, a)] \\ &= g(f(x, a), b). \end{aligned}$$

On the other hand,

$$\begin{aligned} g(f(x, a), b) &= g(C_1[x], b) \\ &= (\gamma\Box.g(C_1[\Box], b))[x] \\ &= (\gamma\Box.g(f(\Box, a), b))[x]. \end{aligned}$$

Therefore, the context $\gamma\Box.g(f(\Box, a), b))$ is often called the composition *of C_1 and C_2, denoted as $C_1 \circ C_2$.*

Example 65. *Suppose f is a binary symbol with all its two arguments active. Suppose C_1 and C_2 are two contexts and a, b are constants. Then easily we get*

$$\begin{aligned} f(C_1[a], C_2[b]) &= (\gamma\Box_2.f(C_1[a], C_2[\Box_2]))[b] \\ &= (\gamma\Box_2.((\gamma\Box_1.f(C_1[\Box_1], C_2[\Box_2]))[a]))[b]. \end{aligned}$$

What happens above is similar to curring *a function that takes two arguments. It says that there exists a context C_a, related with C_1, C_2, f and a of course, such that $C_a[b]$ returns $f(C_1[a], C_2[b])$. The context C_a has a binding hole $\Box_2$, and a body that itself is another context C'_a applied to a. In other words, there exists C_a and C'_a such that*

- $f(C_1[a], C_2[b]) = C_a[b]$,
- $C_a = \gamma\Box_2.(C'_a[a])$,
- $C'_a = \gamma\Box_1.f(C_1[\Box_1], C_2[\Box_2])$.

A natural question is whether there is a context C such that $C[a][b] = f(C_1[a], C_2[b])$.

Proposition 66. *$C_1[C_2[\varphi]] = C[\varphi]$, where $C = \gamma\Box.C_1[C_2[\Box]]$.*

3.3.1 Normal forms

In this section, we consider *decomposition* of patterns. A decomposition of a pattern P is a pair $\langle C, R\rangle$ such that $C[R] = P$. Let us now consider patterns that do not have logical connectives.

3.4 Separation logic and their variants

In this section we show how various variants of separation logic are instances of matching logic. We will choose n most popular variants $\mathrm{SL}_1, \ldots, \mathrm{SL}_n$, and for each of them prove the following conservative extension result:

$$\vDash_{\mathrm{SL}_i} \varphi \quad \text{iff} \quad T_{\mathrm{SL}_i} \vDash t_i(\varphi),$$

where T_{SL_i} is the matching logic theory of SL_i and t_i is the corresponding "separation logic formulae to matching logic patterns" translation function, which in most cases should be the identity function.

3.4.1 Original separation logic

3.4.2 Separation logic with recursive definitions

Since 2005 when Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn firstly proposed separation logic as a formalization of mutable data structures on heaps in the canonical paper [1], many people have studied and proposed various variants of separation logic. Among all variants, the one that was proposed in [2], named *separation logic with recursive/inductive definitions* (SLRD or SLID), seem to be the most popular one [3][4][5][6][7]. In 2014, David Cok initiated the first edition of separation logic competition (SL-COMP) alongside the satisfiability modulo theories competition (SMT-COMP), and successfully held the second edition in 2015. Six state-of-the-art separation logic provers participated in the events and competed on a set of benchmarks [8]. The benchmarks include hundreds of separation logic formulas of fragments of SLRD, namely SLID+, SLNL, and SLL, in which SLID+ considers inductive definitions of the general form given in equation and its satisfiability problem is decidable but the validity of an entailment is not; SLNL is obtained from SLID+ by mainly restricting the inductive definitions to nested lists data structures, and including skip-lists; SLL is a sub-fragment of SLL+ in which only the *ls* definition is used, it's obtained by requiring that both formulas of the entailment problem do not contain disjunctions and the right-hand side formula has no existential quantification.

The main goal of this section is to show that SLRD is an instance of matching logic. We will introduce a matching logic theory T_{SLRD} that is a *conservative extension* of SLRD, which means that every separation logic formula is a well-formed pattern in T_{SLRD} (modulo syntactic sugar), and

$$\vdash_{\mathrm{SL}} \varphi \quad \text{iff} \quad T_{\mathrm{SLRD}} \vdash \varphi \quad \text{for any separation logic formula } \varphi.$$

Separation logic assertions are often written in an extended SMT-LIB format [9]. Therefore we also implemented a code-translator that converts separation logic assertions to matching logic definitions, which can later be fed to the matching logic prover.

Here we only provide a brief review of SLRD. Readers are refered to [2] for more details.

Let $\mathbb{F}$ be a finite nonempty set of field names, $\mathbb{D}$ be a set of data types, and $\mathbb{P}$ be a

finite set of recursive predicates. The next figure summaries the syntax of SLRD.

$f \in \mathbb{F}$ field names	$P \in \mathbb{P}$ recursive definition name
$x, y \in Vars$ reference program vars	$X, Y \in LVars$ reference logical vars
$d, D \in DVars$ data variables	Δ data constraints
$E, F ::= x \mid X$	reference variables
$\rho ::= \{(f, E)\} \mid \{(f, D)\} \mid \rho \cup \rho$	set of field references
$\Pi ::= E = F \mid E \neq F \mid \Pi \wedge \Pi$	pure formulas
$\Sigma ::= emp \mid junk \mid E \mapsto \rho \mid (P(\mathbf{E}, \mathbf{D})) \mid \Sigma * \Sigma$	spatial formulas
$A, B \stackrel{\Delta}{=} \exists \mathbf{X}, \mathbf{D}.\Pi \wedge \Sigma \wedge \Delta$	formulas

SLRD describes `states` consisting of a stack and a heap. Let *Loc* be a set of locations and *Val* a set of data values. Let *DVars* be a set of data variables, *LVars* a set of reference logical variables and *Vars* a set of reference program variables. A stack *S* is a function mapping reference variables to non-addressable values and locations, which are disjoint sets. Note that *DVars* maps to *Val*, and *LVars* as well as *Vars* maps to *Loc*. A heap *H* is a partial function that defines values of fields for some of the locations in *Loc*, it can also be seen as a partial mapping from locations to records. The records are defined by the user as a set of fields typed as reference or data. The domain of the heap *H* is denoted by *dom(H)* and the set of locations in the domain of *H* is denoted by *ldom(H)*. Two heaps are disjoint if their domains do not overlap. Symbol *nil* is interpreted to a location $S(nil) \notin ldom(H)$. Their definitions are as follows.

$$\text{Heaps} \stackrel{\text{def}}{=} \text{Loc} \times \text{Fields} \rightharpoonup \text{Val} \cup \text{Loc}$$

$$\text{Stacks} \stackrel{\text{def}}{=} \text{Var} \rightarrow \text{Val} \cup \text{Loc}$$

Let's look at the definitions of SLRD's syntax above. The set of field names $\mathbb{F}$ is defined using the *Field* sort in the theory. Each field is declared as a function symbol of arity 0 and result type *Field A B*. Let *A* be the sort corresponding to the record type declaring the field and *B* be the sort typing the field. The following is an example of the definition of a binary tree node.

```
typedef struct btree_s {
struct btree_s* lson;
struct btree_s* rson;
int data;
}* btree_t;

(declare-fun lson () (Field Btree_t Btree_t))
(declare-fun rson () (Field Btree_t Btree_t))
(declare-fun data () (Field Btree_t Int))
```

Let record values be denoted by mappings of record fields to locations stored in variables. The following is an example value for a record defining a binary tree node.

```
{(lson, x), (rson, y),(data, z)}
```

The set of recursive definitions $\mathbb{P}$ is defined by the syntax $P(\mathbf{E},\mathbf{D}) \stackrel{\Delta}{=} \bigvee_i \exists \mathbf{X_i}, \mathbf{D_i}$, $\Pi_i \wedge \Sigma_i \wedge \Delta_i$ where the spatial formulas Σ_i may call P or other predicates from $\mathbb{P}$. In spatial formulas, we have *emp* that denotes empty heap. We have *junk* that is true for any heap, whether empty or consisting of some allocated nodes. We have * as the separating conjunction symbol.

After we have known the syntax of SLRD, we then turn our attention to the semantics. The logic allows to prove judgments of the form $(S, H) \vDash \varphi$, where S is a stack, H is a heap, and φ is an assertion over the given stack and heap. The semantics are given as follows:

$(S, H) \models E = F$	iff $S(E) = S(F)$
$(S, H) \models E \neq F$	iff $S(E) \neq S(F)$
$(S, H) \models \varphi \wedge \psi$	iff $(S, H) \models \phi$ and $(S, H) \models \psi$
$(S, H) \models emp$	iff $dom(H) = \emptyset$
$(S, H) \models junk$	always
$(S, H) \models E \mapsto \rho$	iff $dom(H) = \{(S(E), f_i) \mid (f_i, E_i) \in \rho\}$
	and for every $(f_i, E_i) \in \rho, H(S(E), f_i) = S(E_i)$
$(S, H) \models \Sigma_1 * \Sigma_2$	iff $\exists H_1, H_2$ s.t. $ldom(H) = ldom(H_1) \uplus ldom(H_2)$,
	$(S, H_1) \models \Sigma_1,$ and $(S, H_2) \models \Sigma_2$
$(S, H) \models P(\mathbf{E})$	iff $(S, H) \in [\mathbb{P}](P(\mathbf{E}))$
$(S, H) \models \exists X.\varphi$	iff there exists $l \in Loc$ s.t.$(S[X \leftarrow l], H) \models \varphi$

In the above semantics, $\uplus$ denotes the disjoint union of sets. $S[X \leftarrow l]$ denotes the function S' s.t. $S'(X) = l$ and $S'(Y) = S(Y)$ for any $Y \neq X$. Note that a configuration (S, H) satisfies a predicate atom $P(\mathbf{E})$ if it belongs to the least fix point of the set of recursive definitions $\mathbb{P}$ for the actual parameters $\mathbf{E}$ of P. The set of models of a formula φ is denoted by $[\varphi]$. Given two formulas φ_1 and φ_2, we have $\varphi_1 \Rightarrow \varphi_2$ iff $[\varphi_1] \subseteq [\varphi_2]$.

Now we have got the general idea of SLRD. We will then focus on building a separation logic theory of this variant using matching logic, in other words, to contribute to proving that SLRD is an instance of matching logic. The theory T_{SL} consists of two parts: S is the set of sorts and Σ is the set of symbols. Our definition focus on the *symbolic heaps* part, because there is no need to redefine recursive definitions in matching logic. In the following definition, we can define different data types in $\mathbb{D}$ such as *Nat*, *node*, etc. Let H, H_1, H_2, H_3 be of sort *Map*. Let r, r_1, r_2, r_3 be of sort *Record*. Let *x,y,z* be variables of the same data type. $f \in \mathbb{F}$. $D \in \mathbb{D}$. Let $\mathbb{F}_{\text{D}}$ denote the subset of field

names which take a value of type D to form a *Record*.

$$
\begin{aligned}
&T_{\mathrm{SL}} = \{S, \Sigma\} \\
&S = \{Record, Map\} \cup \mathbb{D} \\
&\Sigma = \{\cup, nil, emp, \mapsto, *, ls\} \cup \mathbb{F} \\
&\quad nil :\rightarrow D && emp :\rightarrow Map \\
&\quad \_ * \_ : Map \times Map \rightharpoonup Map && H_1 * H_2 = H_2 * H_1 \\
&\quad (H_1 * H_2) * H_3 = H_1 * (H_2 * H_3) && emp * H = H \\
&\quad \_ \mapsto \_ : D \times Record \rightharpoonup Map && x \mapsto r_1 * x \mapsto r_2 = \bot \\
&\quad \{(f\ \_)\} : D \rightarrow Record \ \text{ for any } f \in \mathbb{F}_{\mathrm{D}} && nil \mapsto r = \bot \\
&\quad \_ \cup \_ : Record \times Record \rightharpoonup Record && \{(f\ x)\} \cup \{(f\ y)\} = \bot \\
&\quad (r_1 \cup r_2) \cup r_3 = r_1 \cup (r_2 \cup r_3) && r_1 \cup r_2 = r_2 \cup r_1 \\
&\quad ls(x\ z) = (x = z) \vee (\exists y\ (x \neq z \wedge x \mapsto \{(f\ y)\} * ls(y\ z))
\end{aligned}
$$

In S we have sorts *Record*, *Map* and all the data types. In the benchmarks of SL-COMP they have data types *Sll_t*, *TLL_t*, etc. We can define all of them in S. A field reference is the combination of a field name and a data. Let *Record* be a set of field references. Let *Map* be the mapping by symbol $\mapsto$ from a data to a *Record*.

In Σ, the symbol $\cup$ is used to group two *Record* together to form a new *Record*. Note that its semantic is not like *or* because it doesn't conform to the $(\vee)$-rule: $\sigma(..., P \vee Q, ...) = \sigma(..., P, ...) \vee \sigma(..., Q, ...)$. In general it's more like *and*. Let's look at an example: $(S, H) \models E \mapsto \{(lson, 3)\} \cup \{(rson, 4)\}$ iff the domain of heap H is the fields *lson* and *rson* of E's location, and the corresponding values of the fields are 3 and 4. The values of other fields are not concerned. What's more, $\{(lson, 3)\} \cup \{(lson, 4)\}$ can not be satisfied because the function H cannot have multiple values. We use $\cup$ here because we want to keep consistent with the separation logic syntax. Let the symbol *nil* be a program variable representing an undefined reference. Let the symbol *emp* be an empty heap. Symbol $\mapsto$ has been explained above. Let symbol $*$ be the separating conjunction operator. Note that $\cup$ and $*$ are both associative and commutative. Symbol *ls* means list, its definition can be seen above. We define all the field names to be symbols. They take a certain type data as input and form a field reference. The following is an example of a binary tree node.

```
D        : t
Record   : {(lson x), (rson y), (data z)}
Map      : t |-> {(lson x), (rson y), (data z)}
```

In addition, we define a syntactic sugar in our theory called *junk*. It stands for the top pattern $(\top)$, which means total set in matching logic.

$$junk \equiv \top$$

Now we can do the formula transformation from separation logic with recursive definitions to matching logic. We are happy to tell you that all the formulas stay the same in matching logic as in this separation logic variant, so no extra work needed.

Finally, we show an example of translation from the benchmark of SL-COMP to matching logic. In the following separation logic example, data type *Sll_t* is defined. Let *next* be a field name with record type *Sll_t* and field type *Sll_t*. Let x_1 - x_{10} be constants of data type *Sll_t*. The following formula is asserted:

$$((ls\ x_6\ x_1) * (x_4 \mapsto \{(next\ x_9)\}) * emp)$$

The following is the example encoded in the separation logic theory in SMT_LIB. The sorts and the set of reference variables (program or existentially quantified) are declared in a classic way. For example:

```
(declare-sort Btree_t () 0)
(declare-fun root () Btree_t)
```

declares a binary tree sort with no parameters and a variable root to be a reference to a binary tree. The way to declare field names has been explained above. This is an extension from SMT_LIB. Note that in the benchmarks of SL-COMP the flatness of formulas is not ensured. So the spatial atoms are typed in the theory by the *Space* sort (arity 0) and their combination with a pure (boolean) formula requires to cast *Space* to *Bool*. The space atoms are built using the following theory operators.

Syntax	SMT_LIB notation
emp	`emp`
junk	`junk`
$\Sigma_1 * ... * \Sigma_n$	(`ssep` < form >$^+$)
$E \mapsto \rho$	(`pto` < var > ρ)
none	(`tobool` < form > < form >)
none	(`tospace` < form > < form >)
$\{(f, E)\}$	(`ref` < f > < var >)
$\rho \cup \rho$	(`sref` < ρ >$^+$)

In the following codes, note that we only have one field here, so `ref` and `sref` may look redundant, but if we have multiple fields then their differences can be seen.

```
;Sll_t is of arity 0
(declare-sort Sll_t 0)
;next is a field name
(declare-fun next () (Field Sll_t Sll_t))

;List
(define-fun ls ((?in Sll_t) (?out Sll_t)) Space
(tospace (or (= ?in ?out)
(exists ((?u Sll_t))
(and (distinct ?in ?out) (tobool
(ssep (pto ?in (sref (ref next ?u))) (ls ?u ?out)
)))))))

;Sll_t constants
```

```
(declare-fun x0 () Sll_t)
(declare-fun x1 () Sll_t)
(declare-fun x2 () Sll_t)
(declare-fun x3 () Sll_t)
(declare-fun x4 () Sll_t)
(declare-fun x5 () Sll_t)
(declare-fun x6 () Sll_t)
(declare-fun x7 () Sll_t)
(declare-fun x8 () Sll_t)
(declare-fun x9 () Sll_t)
(declare-fun x10 () Sll_t)

;example 1
;((ls x6 x1)*(x4|->{(next x9)})*emp)
(assert
    (tobool
    (ssep
        (ls  x6 x1)
        (pto x4 (ref next x9))
        emp
))
)
(check-sat)
```

The following is the matching logic version of the above example. It shall be noted that proving assertion *P* is equivalent to establishing *not(floor P)* is unsat. Its syntax refers to the matching logic syntax used in section 2.3.

```
(declare-sort Sll_t)
(declare-sort Record)
(declare-sort Map)

;Sll_t constants
(declare-func x0 () Sll_t)
(declare-func x1 () Sll_t)
(declare-func x2 () Sll_t)
(declare-func x3 () Sll_t)
(declare-func x4 () Sll_t)
(declare-func x5 () Sll_t)
(declare-func x6 () Sll_t)
(declare-func x7 () Sll_t)
(declare-func x8 () Sll_t)
(declare-func x9 () Sll_t)
(declare-func x10 () Sll_t)

(declare-func nil () Sll_t)
```

```
;distinct : a syntactic sugar
;distinct(x y) = (not (= x y))

;Field name
(declare-func next (Sll_t) Record)

;Record
;(declare-func ref (Record) Record)

;Union
(declare-part uni (Record Record) Record)

; Maps
(declare-func emp () Map)

; x |-> y
(declare-part pto (Sll_t Record) Map)

; nil |-> y is bottom
(assert (forall ((x Record))
  (not (pto nil x))))

; ssep is separating conjunction
(declare-part ssep (Map Map) Map)

; commutativity
(assert (forall ((h1 Map) (h2 Map))
  (= (ssep h1 h2) (ssep h2 h1))))

; associativity
(assert (forall ((h1 Map) (h2 Map) (h3 Map))
  (= (ssep (ssep h1 h2) h3)
     (ssep h1 (ssep h2 h3)))))

; identity
(assert (forall ((h Map))
  (= h (ssep h emp))))

; x |-> y * x |-> z = bottom
(assert (forall ((x Sll_t) (y Record) (z Record))
  (not (ssep (pto x y) (pto x z)))))

;(next x) U (next y) = bottom
(assert (forall ((x Sll_t) (y Sll_t))
  (not (uni (next x) (next y)))))
```

```
; commutativity
(assert (forall ((r1 Record) (r2 Record))
  (= (uni r1 r2) (uni r2 r1))))

; associativity
(assert (forall ((r1 Record) (r2 Record) (r3 Record))
  (= (uni (uni r1 r2) r3)
     (uni r1 (uni r2 r3)))))

;List
(declare-func ls (Sll_t Sll_t) Map)
(assert (forall ((in Sll_t)(out Sll_t))
 (= (ls in out) (or (= in out) (exists ((u Sll_t)) (and (not
 (= in out)) (ssep (pto in (next u)) (ls u out)
)))))
))

;example 1
;((ls x6 x1)*(x4|->{(next x9)})*emp)
(assert
    (not (floor
        (ssep
          (ls  x6 x1)
        (ssep
          (pto x4 (next x9))
          emp
  ))))
)
```

4 Semantics of K

Moved to a separate repository.

References

[1] Berdine J, Calcagno C, O'Hearn P W. Symbolic execution with separation logic[C]//APLAS. 2005, 5(3780): 52-68.

[2] Radu Iosif, Adam Rogalewicz, and Jiri Simcek. The tree width of separation logic with recursive definitions. In CADE, volume 7898 of *Lecture Notes in Computer Science*, pages 21?38. Springer, 2013.

[3] Berdine J, Calcagno C, O?hearn P W. Smallfoot: Modular automatic assertion checking with separation logic[C]//International Symposium on Formal Methods for Components and Objects. Springer, Berlin, Heidelberg, 2005: 115-137.

[4] Distefano D, O?Hearn P, Yang H. A local shape analysis based on separation logic[J]. Tools and Algorithms for the Construction and Analysis of Systems, 2006: 287-302.

[5] Jacobs B, Piessens F. The VeriFast program verifier[J]. 2008.

[6] Qiu X, Garg P, ?tefanescu A, et al. Natural proofs for structure, data, and separation[J]. ACM SIGPLAN Notices, 2013, 48(6): 231-242.

[7] Sighireanu M, Cok D R. Report on sl-comp 2014[J]. *Journal on Satisfiability, Boolean Modeling and Computation*, 2016, 9: 173-186.

[8] https://github.com/mihasighi/smtcomp14-sl

[9] https://www.irif.fr/ sighirea/slcomp14/smtlib2sl-v0.pdf