Foundations for Language-independent Verification

ANONYMOUS AUTHOR(S)

Text of abstract ....

CCS Concepts: • **Software and its engineering → General programming languages**; • **Social and professional topics** → *History of programming languages*;

Additional Key Words and Phrases: keyword1, keyword2, keyword3

ACM Reference Format:
Anonymous Author(s). 2018. Foundations for Language-independent Verification. *Proc. ACM Program. Lang.* 1, POPL, Article 1 (January 2018), 31 pages.

1 INTRODUCTION AND MOTIVATION (3 PAGES)

We can have a long section about introduction and motivation, running for 2–3 pages. The idea is to emphasize language-independent verification, and say that $\mathbb{K}$ is the best effort for doing that. And then we can say this paper provides foundations for not just $\mathbb{K}$, but language-independent verification in general; or even more: we could claim this paper is for language-independent approach in general, including execution (lambda calculus and contexts, etc.) and verification (reachability logic etc.), model checking (LTL etc.), and more. We should go for 2 pages if we only talk about motivation. We could go for 3 pages if we briefly talk about concrete results and our approach.

Program verification asks if a program satisfies its formal specification. Traditionally, program verification is powered by a program logic such as Hoare logic, dynamic logic, or separation logic, and a set of proof rules are used to define the semantics of the target languages. These language-specific program logics are often hard to design and understand, non-executable, and error-prone, and they do not easily adapt to language change. An alternative approach is to translate target languages to some intermediate verification languages (IVLs) such as Boogie and Why, and verification tools are developed for IVLs in separation from target languages. However, correct translation is hard and can miss behavior detail, resulting in false alarm in state-of-the-art verification tools.

Language-independent verification, in contrary, considers a language framework where any language is given a formal executable semantics and all tools for that language are automatically generated by the framework in a correct-by-construction manner. The $\mathbb{K}$ framework is such an language-independent framework, and many languages have been defined a formal semantics, including C, Java, and JavaScript.

The goal of this paper is to provide foundations for a collection that is currently used in the language-independent framework $\mathbb{K}$. In particular, we want to show that language-independent verification is all about searching for proofs, and $\mathbb{K}$ is a best-effort implementation of such a proof search. Since we consider verification of any properties written in any logics of any programs written in any languages, we need an expressive "unified" logic, which subsumes all popular program logics that are used for verification and "static" logics that are used to state program properties. Previous work has shown that matching logic ....

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

2 MATCHING LOGIC PRELIMINARIES AND EXISTING RESULTS (2 PAGES)

In this section, we introduce the syntax and semantics of matching logic, and present some existing results about its expressive power. In particular, we show how partial functions, first-order logic with equality, modal logic S5, and separation logic can be represented in matching logic as theories or notations. We conclude the section with a preliminary sound and complete proof system.

2.1 Syntax, semantics, and notations

A signature is a triple $\mathbb{\Sigma} = (\textsc{Name}, S, \Sigma)$, where $\textsc{Name}$ is a countably infinite set of variable names, S is an nonempty countable (finite or infinite) sort set, and $\Sigma = \{\Sigma_{s_1 \dots s_n, s}\}_{s_1, \dots, s_n, s \in S}$ is an $(S^* \times S)$-indexed countable (finite or infinite) symbol set. We write $\Sigma_{s_1 \dots s_n, s}$ as $\Sigma_{\lambda, s}$ if $n = 0$. When $\textsc{Name}$ is clear from the context, we often omit it and just write the signature $\mathbb{\Sigma} = (S, \Sigma)$. If S is also clear, we refer to a signature by just Σ. The set of all $\mathbb{\Sigma}$-patterns of sort s, denoted as $\textsc{Pattern}_s(\mathbb{\Sigma})$ or just $\textsc{Pattern}_s$ when the signature is clear from the context, is defined by the following grammar:

$$
\begin{array}{rcll}
\varphi_s & ::= & x{:}s \text{ with } x \in \textsc{Name} & \text{// Variable} \\
& | & \sigma(\varphi_{s_1}, \dots, \varphi_{s_n}) \text{ with } \sigma \in \Sigma_{s_1 \dots s_n, s} & \text{// Structure} \\
& | & \neg \varphi_s & \text{// Complement} \\
& | & \varphi_s \wedge \varphi_s & \text{// Intersection} \\
& | & \exists x{:}s'.\varphi_s \text{ with } s' \in S \text{ (no need be the same as } s) & \text{// Binding}
\end{array}
$$

The set of all variables of sort s is denoted as $\textsc{Var}_s$. Let $\textsc{Var} = \{\textsc{Var}_s\}_{s \in S}$ and $\textsc{Pattern} = \{\textsc{Pattern}_s\}_{s \in S}$ be the S-indexed family set of all variable and patterns. For simpler notation, we often blur the distinction between a family of sets and their union, and use $\textsc{Var}$ and $\textsc{Pattern}$ to denote the set of all variables and patterns respectively. We write $\varphi \in \textsc{Pattern}$ to mean that φ is a pattern, and $\varphi_s \in \textsc{Pattern}$ or $\varphi \in \textsc{Pattern}_s$ to mean that it has sort s. Similarly, $\sigma \in \Sigma$ means σ is a symbol. If $\sigma \in \Sigma_{\lambda, s}$, we say σ is a constant symbol of sort s, and we write σ instead of $\sigma()$. We often drop the sort when writing variables, so instead of $x{:}s$, we just write x. We can define the conventional notions of free variables and alpha-renaming as in first-order logic. We write $\varphi[\psi/x]$ for variable-capture-free substitution, in which alpha-renaming happens implicitly to prevent free variable capturing.

A $\mathbb{\Sigma}$-model (or simply a model) is a pair $\mathcal{M} = (M, \_{}_{\mathcal{M}})$ where $M = \{M_s\}_{s \in S}$ is an S-indexed family of nonempty carrier sets and $\_{}_{\mathcal{M}} = \{\sigma_{\mathcal{M}}\}_{\sigma \in \Sigma}$ maps every symbol $\sigma \in \Sigma_{s_1, \dots, s_n, s}$ to a function $\sigma_{\mathcal{M}} \colon M_{s_1} \times \cdots \times M_{s_n} \to 2^{M_s}$, where 2^{M_s} means the set of all subsets of M_s. In particular, each constant symbol $\sigma \in \Sigma_{\lambda, s}$ maps to a subset $\sigma_{\mathcal{M}} \subseteq M_s$. An $\mathcal{M}$-valuation (or simply a valuation) is a mapping $\rho \colon \textsc{Var} \to M$ such that $\rho(x{:}s) \in M_s$ for every sort $s \in S$. Two valuations ρ_1 and ρ_2 are x-equivalent for some variable x, denoted as $\rho_1 \overset{x}{\sim} \rho_2$, if $\rho_1(y) = \rho_2(y)$ for every y distinct from x. A valuation ρ can be extended to a mapping $\bar{\rho} \colon \textsc{Pattern} \to 2^M$ such that $\bar{\rho}(\varphi_s) \subseteq M_s$, in the following inductive way:

- $\bar{\rho}(x) = \{\rho(x)\}$, for every $x \in \textsc{Var}_s$;
- $\bar{\rho}(\sigma(\varphi_1, \dots, \varphi_n)) = \sigma_{\mathcal{M}}(\bar{\rho}(\varphi_1), \dots, \bar{\rho}(\varphi_n))$, for every $\sigma \in \Sigma_{s_1 \dots s_n, s}$ and appropriate $\varphi_1, \dots, \varphi_n$
- $\bar{\rho}(\neg\varphi) = M_s \setminus \bar{\rho}(\varphi)$, for every $\varphi \in \textsc{Pattern}_s$;
- $\bar{\rho}(\varphi_1 \wedge \varphi_2) = \bar{\rho}(\varphi_1) \cap \bar{\rho}(\varphi_2)$, for every φ_1, φ_2 of the same sort;
- $\bar{\rho}(\exists x.\varphi) = \bigcup\{\overline{\rho'}(\varphi) \mid \text{for every } \rho' \overset{x}{\sim} \rho\}$.

(Grigore) We also need to give some intuition for "matching" (the patterns are "matched" by the values in their interpretation, which becomes literal when the model contains terms.

Intuitively, $\bar{\rho}(\varphi)$ is the set of elements that match the pattern φ. Derived constructs are defined as follows for convenience:

$$\top_s \equiv \exists x{:}s.x{:}s \qquad \bot_s \equiv \neg\top_s$$
$$\varphi_1 \vee \varphi_2 \equiv \neg(\neg\varphi_1 \wedge \neg\varphi_2) \qquad \varphi_1 \rightarrow \varphi_2 \equiv \neg\varphi_1 \vee \varphi_2$$
$$\varphi_1 \leftrightarrow \varphi_2 \equiv (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1) \qquad \forall x.\varphi \equiv \neg(\exists x.\neg\varphi)$$

Interested readers are encouraged to prove these derived constructs have the intended semantics, or refer to [?] for details. We often drop the sort subscripts when there is no confusion.

Given a model $\mathcal{M}$ and a valuation ρ, we say $\mathcal{M}$ and ρ satisfy a pattern φ_s, denoted as $\mathcal{M}, \rho \models \varphi_s$, if $\bar{\rho}(\varphi) = M_s$. We say $\mathcal{M}$ satisfies φ_s or φ_s holds in $\mathcal{M}$, denoted as $\mathcal{M} \models \varphi_s$, if $\mathcal{M}, \rho \models \varphi_s$ for every valuation ρ. We say φ_s is valid if it holds in every model. Let Γ be a pattern set. We say $\mathcal{M}$ satisfies Γ, if $\mathcal{M} \models \varphi$ for every $\varphi \in \Gamma$. We say Γ semantically entails φ, denoted as $\Gamma \models \varphi$, if for every model $\mathcal{M}$ such that $\mathcal{M} \models \Gamma$, $\mathcal{M} \models \varphi$. When Γ is the empty set, we abbreviate $\emptyset \models \varphi$ as just $\models \varphi$, which is equivalent to say φ is valid.

(Xiaohong) Say that we use mathcal fonts $\mathcal{M}$, $\mathcal{I}$, ...to denote models, and the corresponding M, I,...to denote carrier sets. Use capital Greek letters $\Gamma, \Delta, T, \ldots$ to denote pattern set. In particular, use $\mathcal{I}$ to denote intended models and $\mathcal{S}$ to denote standard models.

Given a signature Σ, matching logic gives us all Σ-models. Sometimes, we are only interested in some models, instead of all models. There are typically two ways to restrict models. One way is to define a syntactic matching logic theory (Σ, H) where H is a set of patterns called axioms. A model $\mathcal{M}$ belongs to the theory (Σ, H) if $M \models H$. Syntactic theories are preferred if the models of interest can be axiomatized by a recursively enumerable set H. Most syntactic theories defined in this paper have finite axiom sets. Alternatively, we can define a semantic matching logic theory (Σ, C) where C is a collection of Σ-models. A model $\mathcal{M}$ belongs to the theory (Σ, C) if $\mathcal{M} \models \varphi$ for all φ that holds in all models in C. Usually, the collection C is a singleton set containing exactly one model $\mathcal{I}$, often referred as the intended model or the standard model, and we abbreviate $(\Sigma, \{\mathcal{I}\})$ as $(\Sigma, \mathcal{I})$. Semantic theories are preferred if the intended model is not axiomatized by any recursively enumerable set of axioms; this is often the case for initial algebra semantics or domains which are defined by induction, such as natural numbers (with multiplication) and finite maps. We will see an example of semantic theories in Section 2.2.4.

Syntactic theories support a notion of proofs (see Section 3) as they have an axiom set, but semantic theories offer arbitrary expressiveness. They both are means to restricting models, and we simply say "theories" when their distinction is not the emphasis.

2.2 Known results about matching logic expressiveness power

(Grigore) I think it is also important to state that when all models are assumed, then ML has been shown to have the same expressiveness as FOL=, but like it is the case with capturing SL, or in FOL with induction, or in initial algebra semantics, one can reduce the set of models and thus get arbitrary expressiveness.

A few important logics and calculus are shown to be definable in matching logic, including propositional calculus, predicate logic, algebraic specification, first-order logic with equality, modal logic S5, and separation logic. In this section, we only discuss a few of them and refer interested readers to [?] for more details.

2.2.1 Definedness, equality, membership, functions, and partial functions. As we have seen, patterns are interpreted as sets. When it comes to classical reasoning for existing mathematical domains, we need a way to interpret patterns in a conventional, two-value way; for example, the total set means true and the empty set means false. We also want to lift reasoning with sort s_1 to sort s_2. In matching logic, the above is methodologically achieved by *definedness* symbols. For any two sorts s and s' which need not be distinct, the definedness symbol $\lceil\_\rceil_s^{s'} \in \Sigma_{s,s'}$ is a unary

matching logic symbol which has an axiom $\lceil x{:}s \rceil^{s'}_{s}$ called the definedness axiom. The axioms makes $\lceil \_ \rceil^{s'}_{s}$ behaves like a predicate that checks definedness:

$$\bar{\rho}(\lceil \varphi_s \rceil^{s'}_{s}) = M_{s'} \quad \text{if } \bar{\rho}(\varphi_s) \neq \emptyset \qquad\qquad \bar{\rho}(\lceil \varphi_s \rceil^{s'}_{s}) = \emptyset \quad \text{if } \bar{\rho}(\varphi_s) = \emptyset$$

Definedness symbols allow us to define many useful derived constructs, including

$\lfloor \varphi \rfloor^{s'}_{s} \equiv \neg \lceil \neg \varphi \rceil^{s'}_{s}$ that checks totality $\quad x \in^{s'}_{s} \varphi \equiv \lceil x \wedge \varphi \rceil^{s'}_{s}$ that checks membership

$\varphi_1 =^{s'}_{s} \varphi_2 \equiv \lfloor \varphi_1 \leftrightarrow \varphi_2 \rfloor^{s'}_{s}$ that checks equality $\quad \varphi_1 \subseteq^{s'}_{s} \varphi_2 \equiv \lfloor \varphi_1 \rightarrow \varphi_2 \rfloor^{s'}_{s}$ that checks containment

These derived constructs have the intended semantics. For example,

$$\bar{\rho}(\varphi_1 =^{s'}_{s} \varphi_2) = M_{s'} \quad \text{if } \bar{\rho}(\varphi_1) = \bar{\rho}(\varphi_2) \qquad\qquad \bar{\rho}(\varphi_1 =^{s'}_{s} \varphi_2) = \emptyset \quad \text{if } \bar{\rho}(\varphi_1) \neq \bar{\rho}(\varphi_2)$$

We drop their sort subscripts if there is no confusion.

In matching logic, symbols are interpreted relationally $\sigma_{\mathcal{M}} \colon M_{s_1} \times, \ldots, \times M_{s_n} \rightarrow 2^{M_s}$. Sometimes, we want to state a symbol σ is to be interpreted as a function in all models. This is achieved by adding the axiom $\exists y.\sigma(x_1, \ldots, x_n) = y$ where $x_1, \ldots, x_n, y$ are distinct. Partial functions can be defined in the similar way, by adding the axiom $\neg\sigma(x_1, \ldots, x_n) \vee \exists y.\sigma(x_1, \ldots, x_n) = y$. We point out that partial functions has been the main subject of research in partial first-order logic, with various logics and axioms proposed to capture the desired properties of definedness and undefinedness; while matching logic allows us elegantly define definedness and partial functions, without a need to develop a new logic.

2.2.2 Modal logic S5. One of the most popular modal logics, S5, is definable in matching logic. S5 is parametric on a set of atomic propositions AP, and its syntax, as shown below, extends propositional calculus with a unary modality $\Box$ that captures necessity: $\Box\varphi$ is read as "it is necessary that φ". A dual modality $\Diamond\varphi \equiv \neg(\Box\neg\varphi)$ is defined to capture possibility.

$$\varphi ::= \text{AP} \mid \varphi \rightarrow \varphi \mid \neg\varphi \mid \Box\varphi$$

S5 formulas are interpreted on a set W of worlds with a valuation $v \colon \text{AP} \times W \rightarrow \{true, false\}$ stating that each proposition holds in a given subset of worlds. The valuation is then extended to S5 formulas inductively:

- $v(\neg\varphi, w) = true$ if $v(\varphi, w) = false$; otherwise, $v(\neg\varphi, w) = false$.
- $v(\varphi_1 \rightarrow \varphi_2, w) = true$ if $v(\varphi_1, w) = false$ or $v(\varphi_2, w) = true$; otherwise, $v(\varphi_1 \rightarrow \varphi_2, w) = false$.
- $v(\Box\varphi, w) = true$ if $v(\varphi, w') = true$ for every $w' \in W$; otherwise, $v(\Box\varphi, w) = false$.

An S5-formula is valid, denoted as $\vDash_{S5} \varphi$, if for every world set W and valuation v, $v(\varphi, w) = true$ for every $w \in W$. S5 admits the following sound and complete proof system which can derive all valid S5-formulas:

Proof system of Modal logic S5 extends propositional calculus proof system with the following:

(N) $\dfrac{\varphi}{\Box\varphi}$ (K) $\Box(\varphi_1 \rightarrow \varphi_2) \rightarrow (\Box\varphi_1 \rightarrow \Box\varphi_2)$

(M) $\Box\varphi \rightarrow \varphi$ (5) $\Diamond\varphi \rightarrow \Box\Diamond\varphi$

Modal logic S5 can be faithfully captured by a matching logic theory which we refer to as S5. The theory S5 has a signature $\mathbb{\Sigma} = (S, \Sigma)$ where S contains exactly one sort, say *world*, and Σ contains a unary symbol $\Diamond \in \Sigma_{world, world}$, plus a constant symbol $p \in \Sigma_{\lambda, world}$ for every atomic proposition $p \in$ AP. The theory S5 has only one axiom, $\Diamond w$, which states that the symbol $\Diamond$ is the definedness symbol. The totality $\Box\varphi \equiv \neg(\Diamond\neg\varphi)$ is defined as a derived construct. With the above definitions,

Any S5-formula is a matching logic pattern in the theory S5.

We establish an important result known as the "conservative extension", which is proved via a model-theoretic approach. We elaborate the proof of conservative extension for S5 here as an example, since the same proof idea will show up multiple times in the rest of the paper. In short, we will show there is a one-to-one correspondence between pairs (W, v) of an S5 world set and a valuation, and matching logic models of the theory S5. The correspondence works in two directions. For the direction from S5 to matching logic, the correspondence takes W as the carrier set of sort *world*, and interprets $\Diamond$ as the definedness predicate $\Diamond_{world}(w) = W$ for every $w \in W$. Moreover, every atomic proposition $p \in \mathrm{AP}$ is interpreted to the subset of worlds $p_{world} = \{w \in W \mid v(p, w) = true\}$. The other direction from matching logic to S5 is left for the interested readers. Under this one-to-one correspondence, we can prove by structural induction that for every S5-formula φ,

$$v(\varphi, w) = true \quad \text{if and only if} \quad w \in \bar{\rho}(\varphi).$$

Notice that S5-formulas contain no variables as matching logic patterns, so it does not matter which valuation ρ we pick in the above. Immediately, we know $\vDash_{S5} \varphi$ if and only if $\mathsf{S5} \vDash \varphi$ for every S5-formula φ. By the completeness results of both S5 and matching logic, we know $\vdash_{S5} \varphi$ if and only if $\mathsf{S5} \vdash \varphi$ for every S5-formula φ.

2.2.3 First-order logic and its variants. TBC.

(Xiaohong) This section should say that FOL= and ML have the same expressiveness when considering all models. We may also show the reduction to FOL without equality, or why ML is more expressive than FOL without equality. We may show FOL+lfp here, too.

2.2.4 Separation logic. Separation logic is a logic specifically designed for reasoning about heap structures. The syntax of separation logic extends first-order logic with some heap operations:

$$\varphi ::= (\text{first-order logic syntax}) \mid emp \mid Nat \mapsto Nat \mid \varphi * \varphi \mid \varphi -\!\!* \varphi$$

Let $s \colon \textsc{Var} \rightharpoonup Nat$ be a partial function called a store and $h \colon Nat \rightharpoonup Nat$ be a partial function called a heap. Separation logic formulas are interpreted in the pair (s, h) inductively as follows:

- $(s, h) \vDash_{\mathrm{SL}} \varphi$ if $s \vDash_{\mathrm{FOL}} \varphi$ and φ is a first-order logic formula;
- $(s, h) \vDash_{\mathrm{SL}} emp$ if the domain of h is empty;
- $(s, h) \vDash_{\mathrm{SL}} e_1 \mapsto e_2$ if $\bar{s}(e_1) \neq 0$, the domain of h is $\{\bar{s}(e_1)\}$, and $h(\bar{s}(e_1)) = \bar{s}(e_2)$;
- $(s, h) \vDash_{\mathrm{SL}} \varphi_1 * \varphi_2$ if there exist disjoint h_1 and h_2 such that $h = h_1 * h_2$ and $(s, h_1) \vDash_{\mathrm{SL}} \varphi_1$ and $(s, h_2) \vDash_{\mathrm{SL}} \varphi_2$;
- $(s, h) \vDash_{\mathrm{SL}} \varphi_1 -\!\!* \varphi_2$ if for every h_1 disjoint with h, if $(s, h_1) \vDash_{\mathrm{SL}} \varphi_1$ then $(s, h_1 * h) \vDash_{\mathrm{SL}} \varphi_2$.

A separation logic formula φ is valid, denoted as $\vDash_{\mathrm{SL}} \varphi$, if $(s, h) \vDash_{\mathrm{SL}} \varphi$ for every store s and heap h.

Separation logic is faithfully captured by a matching logic theory we which denote as SL. The theory SL has the signature $\mathbb{\Sigma} = (S, \Sigma)$ where S contains a sort Nat for natural numbers and a sort Map for heaps. The symbol set Σ contains a constant symbol $emp \in \Sigma_{\lambda, Map}$, a binary symbol $* \in \Sigma_{Nat\ Nat, Map}$, and a binary symbol $\mapsto \in \Sigma_{Map\ Map, Map}$. We write $*$ and $\mapsto$ in mixfix form. The separation conjunction is defined as an alias

$$\varphi_1 -\!\!* \varphi_2 \equiv \exists h.(h \wedge \lfloor h * \varphi_1 \rightarrow \varphi_2 \rfloor)$$

With the above definitions,

Any separation logic formula is a matching logic pattern of sort Map in the theory SL.

Unlike in modal logic S5 where the theory S5 is defined with a set of axioms, the theory SL is defined by a particular matching logic model of the above signature known as the intended model or the standard model, denoted as *Map*. The intended model *Map* has the set of natural numbers $\mathbb{N}$

as the carrier set of sort *Nat*, and the set of partial functions $\{h \mid h\colon \mathbb{N} \rightharpoonup \mathbb{N}\}$ as the carrier set of sort *Map*. Symbols *emp*, $*$, and $\mapsto$ are interpreted in *Map* in the intended way. Notice that separation logic formulas contain no free variables of sort *Map* as matching logic patterns, so the valuation of variables of sort *Map* does not matter. In addition, there is a one-to-one correspondence between Valuations of variables of sort *Nat* and separation logic states. Under this correspondence, we can prove by structural induction that

$(s, h) \models_{\mathrm{SL}} \varphi$ if and only if $h \in \bar{\rho}(\varphi)$ // s and ρ conform to the one-to-one correspondence

As a corollary, $\models_{\mathrm{SL}} \varphi$ if and only if $\mathsf{SL} \vDash \varphi$ for every separation logic formula φ.

2.3 A preliminary proof system

In [?], the author studies sound and complete deduction in matching logic and proposes a preliminary proof system. The proof system, shown in Figure 2.3, is proved to be sound and complete, but it contains axioms that use the equality and membership constructs. Therefore, it cannot be used in theories which do not contain definedness symbols. In Section 3, we will replace this preliminary proof system with a new one that is "pure", i.e., one that does not depend on any particular symbols and works for all theories.

A sound and complete proof system of matching logic that depends on definedness symbols.

3 SOUND AND COMPLETE DEDUCTION IN MATCHING LOGIC (2 PAGES)

This section should be about two pages, together with a float figure of the proof system. State the theorem that the preliminary proof system is provable using this new proof system and definedness axioms. State the completeness result.

(Grigore) You also want to state other results that you proved and might be useful (deduction theorem?). You may also want to have some discussion about the proof system, maybe even that it generalizes modal logic proof rules like the Barcan rule, etc. Basically anything that feels interesting about the proof system should be said here, because we do not want the reader to skip this section too quickly. We want to get their attention.

(Xiaohong) Define the symbol context $C[\_]$ because it appears in the proof system. Say why it's different from variable-capture free substitution, preferably using an example.

In this section, we present a proof system of matching logic, which, unlike the one in Figure 2.3, does not depend on the definedness symbols. Soundness and completeness results are given in Theorem **??**, together with a few important meta-theorems, including a form of deduction theorem.

We have introduced variable-capture-free substitution, where alpha-renaming happens implicitly to prevent free variable capturing. For example, $(\exists y.x)[y/x] \equiv \exists z.y$ where the bound variable y is renamed to z to prevent capturing. In this section, we need another kind of substitution, referred as purely-syntactic-substitution, where alpha-renaming is not carried out, and free variables may be captured after substitution.

(Xiaohong) We use $\varphi[\psi/x]$ for variable-capture-free substitution. We will use $C[\varphi]$ as a shorthand to write contexts application $app(C, \varphi)$, where C is a context pattern built using the context binder γ. We need another notation for purely-syntactic-substitution, as the proof system needs it.

A context C is a pattern with a placeholder variable $\Box \in \textsc{Var}$ that has exactly one free occurrence and no bound occurrence in C. For any pattern φ of the same sort as $\Box$, context application $C[\varphi]$ is the result of replacing $\Box$ for φ in C (without any alpha-renaming). Symbol contexts are a special family of contexts defined as follows

- The pattern $\Box$ is a symbol context called the identity context;
- The pattern $\sigma(\varphi_1, \ldots, \varphi_{i-1}, \Box, \varphi_{i+1}, \ldots, \varphi_n)$ is a symbol context. When patterns $\varphi_1, \ldots, \varphi_{i-1}$, $\varphi_{i+1}, \ldots, \varphi_n$ are not of interest, we just write the pattern as $C_{\sigma,i}$ or simply C_σ;
- The pattern of the form $C_{\sigma_1}[C_{\sigma_2}[\ldots C_{\sigma_n}[\Box]\ldots]]$ is a symbol context where $\sigma_1, \ldots, \sigma_n$ are symbols (not necessarily distinct).

(Proposition$_1$)	$\varphi_1 \longrightarrow (\varphi_2 \longrightarrow \varphi_1)$
(Proposition$_2$)	$(\varphi_1 \longrightarrow (\varphi_2 \longrightarrow \varphi_3)) \longrightarrow (\varphi_1 \longrightarrow \varphi_2) \longrightarrow (\varphi_1 \longrightarrow \varphi_3)$
(Proposition$_3$)	$(\neg\varphi_1 \longrightarrow \neg\varphi_2) \longrightarrow (\varphi_2 \longrightarrow \varphi_1)$
(Modus Ponens)	$\dfrac{\varphi_1 \quad \varphi_1 \longrightarrow \varphi_2}{\varphi_2}$
(Variable Substitution)	$\forall x.\varphi \longrightarrow \varphi[y/x]$
($\forall$)	$\forall x.(\varphi_1 \longrightarrow \varphi_2) \longrightarrow (\varphi_1 \longrightarrow \forall x.\varphi_2)$ if $x \notin FV(\varphi_1)$
(Universal Generalization)	$\dfrac{\varphi}{\forall x.\varphi}$
(Propagation$_\bot$)	$C_\sigma[\bot] \longrightarrow \bot$
(Propagation$_\vee$)	$C_\sigma[\varphi_1 \vee \varphi_2] \longrightarrow C_\sigma[\varphi_1] \vee C_\sigma[\varphi_2]$
(Propagation$_\exists$)	$C_\sigma[\exists x.\varphi] \longrightarrow \exists x.C_\sigma[\varphi]$ if $x \notin FV(C_\sigma[\exists x.\varphi])$
(Framing)	$\dfrac{\varphi_1 \longrightarrow \varphi_2}{C_\sigma[\varphi_1] \longrightarrow C_\sigma[\varphi_2]}$
(Existence)	$\exists x.x$
(Singleton Variable)	$\neg(C_1[x \wedge \varphi] \wedge C_2[x \wedge \neg\varphi])$ where C_1 and C_2 are symbol contexts.

Fig. 1. Sound and Complete Proof System of Matching Logic

Our sound and complete proof system of matching logic is shown in Figure 1, with 13 proof rules/axioms, divided in five categories. We regard axioms as rules with no premises. The first four rules form a complete proof system for propositional calculus. The next three rules are borrowed from predicate logic. Propagation rules are specific to matching logic as far as we know. They allow us to propagate reasoning between structures (see Proposition 3.3). They also capture the "element-wise extension" semantics of symbol applications in matching logic, together with the framing rule, whose importance is well-known in philosophy and logic since 1980s [?]. The last two rules are borrowed from modal logic [?], which play an important role in establishing the completeness result.

A Hilbert-style proof of the proof system in Figure 1 is defined in the usual way. Given a pattern set H and a pattern φ, we write $H \vdash \varphi$ if there exists a proof of φ given H as the hypothesis set. We abbreviate $\emptyset \vdash \varphi$ as $\vdash \varphi$. The soundness result, states as the next theorem, is obtained by carrying out induction on the length of the proof of φ.

Theorem 3.1 (Soundness). *Let H be a pattern set and φ be a pattern. Then $H \vdash \varphi$ implies $H \models \varphi$.*

Several important meta-theorems about the proof system can be proved by carrying out structural induction. We summarize these meta-theorems in the following as propositions.

Proposition 3.2 (Frame reasoning). *Let $\sigma \in \Sigma_{s_1,\ldots,s_n,s}$ be a symbol and $\varphi_i, \varphi_i' \in \text{Pattern}_{s_i}$ be patterns of sort s_i for every $1 \leq i \leq n$. Then $H \vdash \varphi_i \longrightarrow \varphi_i'$ for every $1 \leq i \leq n$ implies $H \vdash \sigma(\varphi_1, \ldots, \varphi_n) \longrightarrow \sigma(\varphi_1', \ldots, \varphi_n')$. Let C be a symbol context and φ_1, φ_2 be patterns of appropriate sort. Then $H \vdash \varphi_1 \longrightarrow \varphi_2$ implies $H \vdash C[\varphi_1] \longrightarrow C[\varphi_2]$.*

Proposition 3.2 says that matching logic supports frame reasoning. Notice that frame reasoning allows us to lift the reasoning in sorts $s_1, \ldots, s_n$ to another sort s, using a symbol $\sigma \in \Sigma_{s_1\ldots s_n,s}$. The soundness of frame reasoning is easily obtained by noticing the fact that $\sigma(\varphi_1, \ldots, \varphi_n)$ is interpreted following an element-wise extension style (see Section 2.1) [**?**].

Proposition 3.3. *For any symbol context C and patterns $\varphi_1, \varphi_2, \varphi$ of appropriate sort, the following propositions hold:*

- $\vdash C[\bot] \leftrightarrow \bot$
- $\vdash C[\varphi_1 \vee \varphi_2] \leftrightarrow C[\varphi_1] \vee C[\varphi_2]$
- $\vdash C[\exists x.\varphi] \leftrightarrow \exists x.C[\varphi] \quad$ *if* $x \notin FV(C[\exists x.\varphi])$

The following propositions hold for any pattern set H. They can be proved from the above results by standard propositional reasoning.

- $H \vdash C[\varphi_1 \vee \varphi_2]$ *if and only if* $H \vdash C[\varphi_1] \vee C[\varphi_2]$
- $H \vdash C[\exists x.\varphi]$ *if and only if* $H \vdash \exists x.C[\varphi] \quad$ *if* $x \notin FV(C[\exists x.\varphi])$

Proposition 3.4. *Let H be a pattern set. For any context C (not necessarily symbol context) and any patterns φ_1, φ_2 of appropriate sort, $H \vdash \varphi_1 \leftrightarrow \varphi_2$ implies $H \vdash C[\varphi_1] \leftrightarrow C[\varphi_2]$.*

Matching logic enjoys a form of deduction theorem. We point out that the usual form of deduction theorem does not hold in matching logic. Specifically, if H is a pattern set and φ, ψ are two patterns of the same sort, and $H \cup \{\varphi\} \vdash \psi$, we cannot conclude that $H \vdash \varphi \longrightarrow \psi$, even if φ is a closed pattern. It is easier and more intuitive to see that from a semantic point of view. Suppose H is empty, and let φ be $\neg x$ and ψ be x. Obviously, $\models \neg x \longrightarrow x$ is wrong; however, $\{\neg x\} \models x$ does hold, as there is no model in which $\neg x$ holds, and thus by definition of the semantic entailment, $\{\neg x\} \models \varphi$ for any φ.

We state the deduction theorem in matching logic in its most general form. Notice that the verbose condition about (Universal Generalization) in the theorem is not surprising, as it is necessary in first-order logic, too [**?**].

Theorem 3.5 (Deduction Theorem). *Suppose the signature contains the definedness symbols. Let H be a pattern set that contains the definedness axioms. For any patterns φ and ψ, if $H \cup \{\psi\} \vdash \varphi$, and the proof does not use (Universal Generalization) on variables that occur free in ψ, then $H \vdash \lfloor\psi\rfloor \longrightarrow \varphi$ where $\lfloor\_\rfloor$ is the totality symbol. In particular, if ψ is closed, then $H \cup \{\psi\} \vdash \varphi$ implies $H \vdash \lfloor\psi\rfloor \longrightarrow \varphi$. Notice the converse theorem also holds. Let H be a pattern set and φ and ψ be two patterns. Then $H \vdash \lfloor\psi\rfloor \longrightarrow \varphi$ implies $H \cup \{\psi\} \vdash \varphi$.*

We provide a semantic point of view to Theorem 3.5. Suppose $H \cup \{\psi\} \models \varphi$, and assume ψ and φ are closed patterns for simplicity. This means that for all models $\mathcal{M}$ such that $\mathcal{M} \models H$, either $\mathcal{M} \not\models \psi$ or $\mathcal{M} \models \varphi$. If $\mathcal{M} \not\models \psi$, then ψ is not interpreted as the total set, and thus $\lfloor\psi\rfloor$ is interpreted as the empty set, and thus $\lfloor\psi\rfloor \longrightarrow \varphi$ is interpreted as the total set. If $\mathcal{M} \models \varphi$, then of course $\mathcal{M} \models \lfloor\psi\rfloor \longrightarrow \varphi$. This illustrates that $H \models \lfloor\psi\rfloor \longrightarrow \varphi$. The rigorous proof of Theorem 3.5 is by carrying out induction on the length of the proof of $H \cup \{\psi\} \models \varphi$.

The completeness of the proof system in Figure 1 requires complicated proofs. As a preliminary result as well as an exercise of using the proof system, we first proved all the proof rules of the preliminary proof system in Figure 2.3, using our proof system in Figure 1 and the definedness axioms. This immediately offers us the completeness result when the signature contains definedness symbols, and the hypothesis set contains definedness axioms. However, we know nothing if definedness symbols are not in the signature; a completely different proof is needed for the general completeness result, which we only state here as the next theorem.

Theorem 3.6. *Let φ be a pattern of sort s and H be a set of patterns such that $H \not\models \bot_{s'}$ for every sort s' different from s. Then $H \models \varphi$ implies $H \vdash \varphi$.*

We explain why we need the condition $H \not\models \bot_{s'}$. Consider the following (counter-)example. Let $\mathbb{\Sigma} = (S, \Sigma)$ be a signature where $S = \{s, s'\}$ and $\Sigma = \emptyset$. Let $H = \{H_s, H_{s'}\}$ where $H_s = \{\bot_s\}$ and $H_{s'} = \emptyset$. Obviously, the hypothesis set H has no model, so $H \models \varphi$ holds for any pattern φ. On the other hand, we can prove that $H \vdash \varphi_{s'}$ holds if and only if $\emptyset \vdash \varphi_{s'}$, for every pattern $\varphi_{s'}$ of sort s'. In particular, $H \not\vdash \bot_{s'}$, which contradicts the fact that $H \models \bot_{s'}$.

As far as we know, the condition $H \not\models \bot_{s'}$ in the completeness theorem is unique in matching logic. In many-sorted first-order logic, such an additional condition is not needed, because even though there are multiple sorts for data, all predicates are in the same world. Thus, inconsistency introduced by one sort s will propagate to the other sort s' in the common world of predicates. For example, one can easily show that in many-sorted first-order logic, $\{t \neq_s t\} \vdash s \neq_{s'} s$, even if s and s' are different sorts.

4 REPRESENTING LOGICS AND CALCULI (15 PAGES)

In this section, we give a list of important logics and calculi about program reasoning that can be defined as matching logic theories or notations. The main point is to convince that matching logic is capable of serving as a unified logic for program verification, that allows us to reason about any properties written in any logics, about any programs written in any programming languages.

4.1 Hybrid Modal Logic

As shown in Section 2.2.2, the modal logic S5 is definable in matching logic. In this section, we show that a first-order extension of modal logic, called hybrid modal logic, is also definable in matching logic. In particular, we show that the famous (Barcan) axioms are special cases of rule (Propagation$_\exists$) for unary symbols.

The syntax of hybrid modal logic contains a set SVar of state variables, a set Nom of nominals, and a binder $\forall$ that binds a state variable in a formula:

$$\varphi ::= p \in \text{AP} \mid x \in \text{SVar} \mid i \in \text{Nom} \mid \neg\varphi \mid \varphi \wedge \varphi \mid \Box\varphi \mid \forall x.\varphi \text{ where } x \in \text{SVar}$$

As in first-order logic, we define $\exists x.\varphi \equiv \neg(\forall x.\neg\varphi)$. As in modal logic S5, we define $\Diamond\varphi \equiv \neg(\Box\neg\varphi)$.

A hybrid modal logic model $\mathcal{M} = (S, R, V)$ is a triple where S is a nonempty set of states, R is a binary relation on S, and $V\colon \text{AP} \cup \text{Nom} \to 2^S$ is a valuation. The valuation V is called a standard valuation if $V(i)$ is a singleton set for every $i \in \text{Nom}$. A model is standard if its valuation is standard. An assignment $g\colon \text{SVar} \to 2^S$ is called standard if $g(x)$ is a singleton set for every $x \in \text{SVar}$. The semantics is defined inductively as follows:

- $\mathcal{M}, g, s \models_{hybrid} p$ if $s \in V(p)$ where $p \in \text{AP}$;
- $\mathcal{M}, g, s \models_{hybrid} x$ if $x \in g(x)$ where $x \in \text{SVar}$;
- $\mathcal{M}, g, s \models_{hybrid} i$ if $i \in V(i)$ where $i \in \text{Nom}$;
- $\mathcal{M}, g, s \models_{hybrid} \neg\varphi$ if $\mathcal{M}, g, s \not\models_{hybrid} \varphi$;
- $\mathcal{M}, g, s \models_{hybrid} \varphi_1 \wedge \varphi_2$ if $\mathcal{M}, g, s \models_{hybrid} \varphi_1$ and $\mathcal{M}, g, s \models_{hybrid} \varphi_2$;

- $\mathcal{M}, g, s \vDash_{hybrid} \Box\varphi$ if $\mathcal{M}, g, s' \vDash_{hybrid} \varphi$ for every s' such that sRs';
- $\mathcal{M}, g, s \vDash_{hybrid} \forall x.\varphi$ if $\mathcal{M}, g', s \vDash_{hybrid} \varphi$ for every g' such that $g' \overset{x}{\sim} g$.

A hybrid modal logic formula is valid if $\mathcal{M}, g, s \vDash_{hybrid} \varphi$ holds for any standard model $\mathcal{M}$, standard assignment g, and state s A sound and complete proof system of hybrid modal logic is proposed in [], as shown in the following. We write $\vdash_{hybrid} \varphi$ if φ is provable in hybrid modal logic.

Proof system of hybrid modal logic extends propositional calculus with the following:

(K)	$\Box(\varphi \to \psi) \to (\Box\varphi \to \Box\psi)$	(N)	$\dfrac{\varphi}{\Box\varphi}$
(Q_1)	$\forall x.(\varphi \to \psi) \to (\varphi \to \forall x.\psi)$, if $x \notin FV(\varphi)$	(Gen)	$\dfrac{\varphi}{\forall x.\varphi}$
(Q_2-Svar)	$\forall x.\varphi \to \varphi[y/x]$	(Q_2-Nom)	$\forall x.\varphi \to \varphi[i/x]$
(Barcan)	$\forall x.\Box\varphi \to \Box\forall x.\varphi$		
(Nom)	$\forall x.(\Diamond^m(x \wedge \varphi) \to \Box^n(x \to \varphi))$, for every $m, n \in \mathbb{N}$	(Name)	$\exists x.x$

We can define a matching logic theory hybridML = $(\mathbb{\Sigma}, H)$ that faithfully captures hybrid modal logic. The matching logic signature $\mathbb{\Sigma}$ = (SVar ∪ Nom, $\{state\}, \Sigma$) where the variable set contains all state variables and nominals and the sort set contains exactly one sort *state* of states. The symbol set Σ consists of

- a unary symbol $\Diamond \in \Sigma_{state,state}$; Define the derived construct $\Box\varphi \equiv \neg(\Diamond\neg\varphi)$;
- a constant symbol $p \in \Sigma_{\lambda,state}$ for every atomic proposition $p \in$ AP;

The axiom set $H = \emptyset$. With the above definitions,

Any hybrid modal logic formula φ is a matching logic pattern in theory hybridML.

In addition, the state/valuation models of hybrid modal logic are essentially identical to the matching logic models of theory hybridML, as summarized as follows.

Hybrid modal logic	Matching logic theory hybridML
state set S	carrier set S
binary relation $R \subseteq S \times S$	interpretation of $\Diamond$ such that $s \in \Diamond_{\mathcal{M}}(t)$ if and only if sRt
standard valuation V: AP $\to 2^S$	interpretation of p such that $p_{\mathcal{M}} = V(p)$
standard valuation V: Nom $\to 2^S$	valuation ρ: Nom $\to S$ such that $V(i) = \{\rho(i)\}$
standard valuation g: SVar $\to 2^S$	valuation ρ: SVar $\to S$ such that $g(x) = \{\rho(x)\}$
$\mathcal{M}, g, s \vDash_{hybrid} \varphi$	$s \in \bar{\rho}(\varphi)$

Immediately, we have the following conservative extension result for hybrid modal logic.

Theorem 4.1 (Conservative Extension for Hybrid Modal Logic). *Let φ be a hybrid modal logic formula. Then $\vdash_{hybrid} \varphi$ if and only if $\vDash_{hybrid} \varphi$ if and only if* hybridML $\vDash \varphi$ *if and only if* hybridML $\vdash \varphi$.

We point out that the famous (N) and (Barcan) axioms in hybrid modal logic are in fact special cases of (Propagation$_\bot$) and (Propagation$_\exists$) rules in matching logic where the symbol takes exactly one argument. For any symbol $\sigma \in \Sigma_{s,s}$, define its complement $\bar{\sigma}(\varphi) \equiv \neg\sigma(\neg\varphi)$. We can show the following propositions hold for any pattern set H in matching logic:

- $H \vdash \varphi$ implies $H \vdash \bar{\sigma}(\varphi)$;
- $\emptyset \vdash \forall x\bar{\sigma}(\varphi) \to \bar{\sigma}(\forall x.\varphi)$.

4.2 Polyadic modal logic

...

4.3 Binders and lambda calculus (2 pages)

We show how to define binders in matching logic, and the conservative extension result for lambda calculus, proved via a model-theoretic approach. Give the proof sketch or explain the proof idea, as we have similar proofs for LTL, CTL, etc. The idea of this section is to claim we can do high-order reasoning in matching logic, but we need to be very careful about that, as we know matching logic is first-order. Maybe what we can claim is that matching logic can reason about high-order functions. After we define lambda calculus, we can say program execution is just a proof search in matching logic. Of course, program execution in $\mathbb{K}$ requires more support, including contexts and strictness, etc. But in theory, it's enough, as lambda calculus is Turing complete.

In this section, we show that binders are definable in matching logic and lambda calculus can be defined as a matching logic theory.

Theorem 4.2. *Let t_1 and t_2 be two lambda terms. Then $\vdash_\lambda t_1 = t_2$ if and only if $\Lambda \vdash t_1 = t_2$.*

4.4 Fixpoints (2 pages)

Fixpoints play an important role in program verification and inductive domains. In mathematics, a fixpoint is a solution of equation $x = e$ where x is a variable of some domain M and e is an expression. If the domain M has a partial order $\leq$, we can compare fixpoints, and as a convention, we use $\mu x.e$ and $\nu x.e$ to denote the least and the greatest fixpoints (abbreviated as lfp and gfp). In matching logic, patterns are interpreted to sets. The equation $x = e$ is an equation of sets, and its solution is a set X which makes the equation hold. Notice that we regard the variable x as merely a placeholder. The fixpoint set X needs not be a singleton set.

We define two binders μ and ν as the *fixpoint constructs*. Axioms about μ and ν are given in the following, where x is a variable, e is a pattern where x does not occur negatively, and e' is a pattern:

(Fix$_\mu$)	$\mu x.e = e[\mu x.e/x]$	(Fix$_\nu$)	$\nu x.e = e[\nu x.e/x]$
(Lfp)	$\lfloor e[e'/x] \to e' \rfloor \to (\mu x.e \to e')$	(Gfp)	$\lfloor e' \to e[e'/x] \rfloor \to (e' \to \nu x.e)$

As we will see later, the side condition that x does not occur negatively in e guarantees the existence of lfps and gfps. By simple matching logic reasoning, we can show that lfps and gfps are dual:

$$\nu x.e = \neg(\mu x.(\neg e[\neg x/x])) \qquad \mu x.e = \neg(\nu x.(\neg e[\neg x/x]))$$

In the following, we consider soundness and completeness of our axiomatization of lfps and gfps in matching logic. We will prove soundness, while completeness in general does not hold. However, axioms (Lfp) and (Gfp) offer some sort of inductive reasoning. Our experiment shows that many interesting inductive properties about heaps in separation logic are provable with (Lfp) and (Gfp). Many temporal logics (e.g. LTL, CTL, μ-calculus), dynamic logic, and reachability logic are also definable using (Lfp) and (Gfp).

We recall readers of an important fixpoint theorem known as the Knaster-Tarski theorem. The Knaster-Tarski theorem has many versions, and we adopt the version that fits the best here in our setting.

Theorem 4.3 (Knaster-Tarski). *Given M is a set and $f : 2^M \to 2^M$ is a function. A set X is called a prefixpoint if $F(X) \subseteq X$ and a postfixpoint if $X \subseteq F(X)$. If f is nondecreasing, then f has a least fixpoint that coincides with its least prefixpoint, and a greatest fixpoint that coincides with its greatest postfixpoint.*

(Xiaohong) Use $\mathcal{I}$ to denote intended models.

We define a notion of the *intended interpretation* or *intended semantics* for lfps and gfps. Given a signature Σ containing the definedness symbols and the binders μ and ν. For simplicity, assume

the signature $\mathbb{\Sigma}$ contains exactly one sort. Let $\mathcal{M}$ be a $\mathbb{\Sigma}$-model and M be the carrier set. Let ρ be a valuation. Let e be a pattern where x does not occur negatively, then e defines a function $[\![e]\!]_{\mathcal{M},\rho}\colon 2^M \to 2^M$ such that $[\![e]\!]_{\mathcal{M},\rho}(X)$ is the interpretation of e when x is "evaluated" to the set X and all other variables y is evaluated to $\rho(y)$. Since x does not occur negatively in e, the function $[\![e]\!]_{\mathcal{M},\rho}$ is non-decreasing (with respect to set containment). Thus by Theorem 4.3, it has a lfp denoted as $\mu[\![e]\!]_{\mathcal{M},\rho}$ and a gfp denoted as $\nu[\![e]\!]_{\mathcal{M},\rho}$. We say $\mathcal{M}$ admits the intended interpretation / has the intended semantics / is an intended model, if for every valuation ρ, any variable x and any pattern e such that x does not occur negatively in e,

$$\bar{\rho}(\mu x.e) = \mu[\![e]\!]_{\mathcal{M},\rho} \qquad\qquad \bar{\rho}(\nu x.e) = \nu[\![e]\!]_{\mathcal{M},\rho}$$

Axioms (Fix) + (Lfp) + (Gfp) are *sound* if they hold in intended models, and are *complete* if only intended models satisfy them. As we will see, these axioms for lfps and gfps are sound but not complete. Take lfps as an example. Let $\mathcal{M}$ be an intended model as above, and ρ be a valuation. Axiom (Fix_μ) holds by definition. Axiom (Lfp) holds if for every pattern e' such that $[\![e]\!]_{\mathcal{M},\rho}(\bar{\rho}(e')) \subseteq \bar{\rho}(e')$, then $\mu[\![e]\!]_{\mathcal{M},\rho} \subseteq \bar{\rho}(e')$. Notice the condition says that $\bar{\rho}(e')$ is a prefixpoint of the function $[\![e]\!]_{\mathcal{M},\rho}$. By Theorem 4.3, axiom (Lfp) holds in intended models.

One may think that (Lfp) says exactly that $[\![e]\!]_{\mathcal{M},\rho}$ is the least prefixpoint (and thus the least fixpoint), so all models satisfying (Lfp) must be intended models. That is not true. Axiom (Lfp) is in fact weaker. It only says that $[\![e]\!]_{\mathcal{M},\rho}$ is the least prefixpoint among all prefixpoints $\bar{\rho}(e')$ that are expressible in the logic and witnessed by a pattern e'. When the carrier set M is infinite, the number of all subsets is uncountably infinite, while the number of all patterns is countably infinite. Therefore, almost all subsets are not expressible, and axiom (Lfp) is far from complete.

Axiom (Lfp)	if $[\![e]\!]_{\mathcal{M},\rho}(\bar{\rho}(e')) \subseteq \bar{\rho}(e')$ then $\mu[\![e]\!]_{\mathcal{M},\rho} \subseteq \bar{\rho}(e')$	for all patterns e'.
Intended models	if $[\![e]\!]_{\mathcal{M},\rho}(Y) \subseteq Y$ then $\mu[\![e]\!]_{\mathcal{M},\rho} \subseteq Y$	for all subsets $Y \subseteq M$.

(Xiaohong) Show how to define recursive functions.

(Xiaohong) Show how to do inductive reasoning.

4.5 Transition systems

Transition systems are important in computer science. They are used to model various types of hardware and software systems, and many algorithms and techniques are proposed to analyze transition systems and to reason about their properties. In mathematics, a transition system $\mathcal{M} = (M, A, \{\xrightarrow{a}\}_{a\in A})$ is a carrier set M and a label set A, accompanied with a binary relation $\xrightarrow{a} \subseteq M \times M$ for every label a. When A contains exactly one label, we write $\mathcal{M} = (M, \to)$ and call the transition system *unlabeled*.

This subsection is dedicated to defining transition systems in matching logic. Let us first consider unlabeled transition systems. Given an unlabeled transition system $\mathcal{M} = (M, \to)$. There are two ways to capture the transition relation $\to$. One is to define a predecessor function $pred\colon M \to 2^M$ such that $pred(t) = \{s \mid s \to t\}$. The other is to define a successor function $succ\colon M \to 2^M$ such that $succ(s) = \{t \mid s \to t\}$. It is easy to show that $succ(s) = \{t \mid s \in pred(t)\}$. Define a unary matching logic symbol $\bullet$ and interpret it to the predecessor function $\bullet_{\mathcal{M}}(t) = pred(t)$. We write $\bullet\varphi$ instead of $\bullet(\varphi)$. Define a derived construct $\overline{\bullet}(\varphi) \equiv \exists x.(x \wedge \lfloor \varphi \to \bullet(\varphi) \rfloor)$, and one can prove that $\overline{\bullet}$ is interpreted to the successor function $\overline{\bullet}_{\mathcal{M}}(s) = succ(s)$. We call the symbol $\bullet$ "strong next" and the construct $\overline{\bullet}$ "strong previous", and we will see why. Since "strong next" and "strong previous" are similar, let us only discuss the "strong next" $\bullet$.

Assume ρ is any valuation and $\bar{\rho}\colon \text{Pattern} \to 2^M$ is the corresponding extended valuation that interprets patterns to sets. Let $s \in M$ and φ be a pattern. The symbol $\bullet$ is called "strong next",

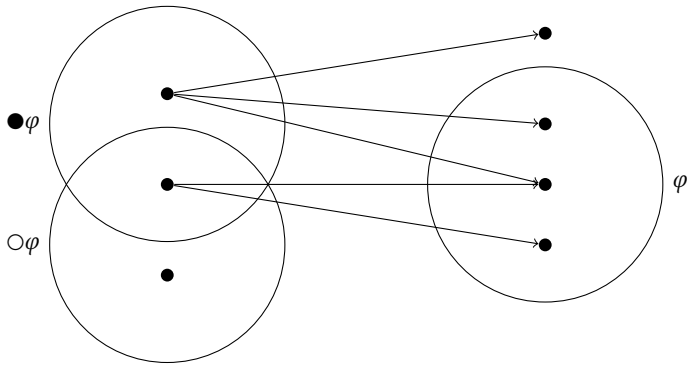

Fig. 2. Strong next ●φ and weak next ○φ.

because $s \in \bar{\rho}(\bullet\varphi)$ if and only if *there exists* $t \in M$ such that $s \longrightarrow t$ and $t \in \bar{\rho}(\varphi)$. In other words, $\bullet\varphi$ holds in the state s if there exists a next state t in which φ holds. We can define a dual construct $\circ\varphi \equiv \neg\bullet\neg\varphi$ called the "weak next", and show that $s \in \bar{\rho}(\circ\varphi)$ if and only if *for all* $t \in M$ such that $s \longrightarrow t$, $t \in \bar{\rho}(\varphi)$. In other words, $\circ\varphi$ holds in the state s if for all next states t, φ holds in t. In particular, if s has no next state, $\circ\varphi$ holds in s unconditionally. Figure 2 illustrates the meaning of $\bullet\varphi$ and $\circ\varphi$.

(Xiaohong) Add a diagram here showing why "strong next" ● is interpreted as the "predecessor function" in models.

"Strong next" ● together with fixpoint constructs μ and ν provide great expressive power about transition systems. The following table summarizes a few important patterns and constructs about transition systems that are used later in the paper. Notice that the right column presents the *intended interpretation* of these patterns and constructs, where μ and ν are interpreted as true lfps and gfps.

Matching logic patterns and constructs	Intended semantics in the transition system $\mathcal{M}$ Necessary and sufficient condition of $s \in \bar{\rho}$(lhs) for some state $s \in M$
$\bullet\varphi$	there exists a state t such that $s \longrightarrow t$ and $t \in \bar{\rho}(\varphi)$
$\circ\varphi$	for all states t such that $s \longrightarrow t$, $t \in \bar{\rho}(\varphi)$
$\bullet\top$	s is a non-terminating state (has a next state)
$\circ\bot$	s is a terminating state (has no next state)
$\mu f.\circ f$	all paths starting at s are finite
$\Diamond\varphi \equiv \mu f.(\varphi \vee \bullet f)$	there exists $n \geq 0$ and $t_1, \ldots, t_n$ such that $s \longrightarrow t_1 \longrightarrow \cdots \longrightarrow t_n$ and $t_n \in \bar{\rho}(\varphi)$
$\Box\varphi \equiv \nu f.(\varphi \wedge \circ f)$	for all $n \geq 0$ and $t_1, \ldots, t_n$ such that $s \longrightarrow t_1 \longrightarrow \cdots \longrightarrow t_n$, $t_n \in \bar{\rho}(\varphi)$
$\mu f.(\varphi_1 \vee (\varphi_2 \wedge \bullet f))$	there exists $n \geq 0$ and $t_1, \ldots, t_n$ such that $s \longrightarrow t_1 \longrightarrow \cdots \longrightarrow t_n$, $t_n \in \bar{\rho}(\varphi_2)$ and $s, t_1, \ldots, t_{n-1} \in \bar{\rho}(\varphi_1)$
$\mu f.(\varphi_1 \vee (\varphi_2 \wedge \circ f))$	for all $n \geq 0$ and $t_1, \ldots, t_n$ such that $s \longrightarrow t_1 \longrightarrow \cdots \longrightarrow t_n$, there exists $m \leq n$ such that $t_n \in \bar{\rho}(\varphi)$ $t_m \in \bar{\rho}(\varphi_2)$ and $s, t_1, \ldots, t_{m-1} \in \bar{\rho}(\varphi_1)$

Using these constructs, we can write axioms to specify properties of transition systems and rule out those which do not satisfy the properties. These axioms allow us to capture various interesting types of transition systems, which we summarize in the next table. Notice that we do not claim these axiomatizations are *complete*, because of the non-standard models of fixpoint constructs. However, we will show in the following subsections that these axioms *completely* capture various important logics for transition systems, including mu-calculus, linear temporal logic (LTL), computation tree logic (CTL), and CTL*.

Matching logic axioms	Transition system $\mathcal{M}$ in the intended semantics
(Inf) $\bullet\top$	Every state in $\mathcal{M}$ has a next state
(Fin) $\mu f.\circ f$	$\mathcal{M}$ has no infinite trace
(Lin) $\bullet\varphi \to \circ\varphi$	Every state in $\mathcal{M}$, if it has next states, has a unique next state; In other words, every state has a linear future

Labeled transition systems can be captured in the similar way. Given a labeled transition system $\mathcal{M} = (M, A, \{\xrightarrow{a}\}_{a\in A})$. Define a matching logic signature Σ which contains a unary symbol $\bullet_a$ for every label $a \in A$.

(Xiaohong) Finish this section.

4.6 Mu-calculus

Mu-calculus is the extension of modal logic with induction and fixpoints. The syntax of mu-calculus is parametric on a set Var of variables, a set AP of atomic propositions, and a set Label of labels. Mu-calculus formulas are defined inductively as follows.

$$\varphi ::= p \in \mathrm{AP} \mid x \in \mathrm{Var} \mid \varphi \wedge \varphi \mid \neg\varphi \mid [a]\varphi \mid \mu x.\varphi \text{ if } x \text{ does not occur negatively in } \varphi$$

where $a \in$ Label. As in matching logic, define $\nu x.\varphi \equiv \neg\mu x.(\neg(\varphi[\neg x/x]))$. Notice that if x does not occur negatively in φ, so does $\neg(\varphi[\neg x/x])$. Define $\langle a\rangle\varphi \equiv \neg[a](\neg\varphi)$.

Mu-calculus formulas are interpreted on structures. A structure $\mathcal{M} = (S, R, V)$ consists of a set S of states, a family set $R = \{R_a\}_{a\in\mathrm{Label}}$ with a binary relation $R_a \subseteq S \times S$ for each label a, and a valuation $V\colon \mathrm{AP} \to 2^S$. Given a structure $\mathcal{M}$ and an assignment $g\colon \mathrm{Var} \to 2^S$, the interpretation function $[\![\_]\!]_{\mathcal{M},g}$ is defined inductively as follows.

- $[\![p]\!]_{\mathcal{M},g} = V(p)$;
- $[\![x]\!]_{\mathcal{M},g} = g(x)$;
- $[\![\varphi_1 \wedge \varphi_2]\!]_{\mathcal{M},g} = [\![\varphi_1]\!]_{\mathcal{M},g} \cap [\![\varphi_2]\!]_{\mathcal{M},g}$;
- $[\![\neg\varphi]\!]_{\mathcal{M},g} = S \setminus [\![\varphi]\!]_{\mathcal{M},g}$;
- $[\![[a]\varphi]\!]_{\mathcal{M},g} = \{s \mid \text{for all } t \text{ such that } sR_at,\ t \in [\![\varphi]\!]_{\mathcal{M},g}\}$;
- $[\![\mu x.\varphi]\!]_{\mathcal{M},g} = \bigcap\{X \subseteq S \mid [\![\varphi]\!]_{\mathcal{M},g'} \subseteq X \text{ where } g' \overset{x}{\sim} g \text{ and } g(x) = X\}$.

A mu-calculus formula φ is valid, written as $\vDash_\mu \varphi$, if $[\![\varphi]\!]_{\mathcal{M},g} = S$ for every structure $\mathcal{M} = (S, R, V)$ and assignment g. Sound and complete deduction for mu-calculus remained open for more than a decade, until Walukiewicz showed in [?] that the following proof system, initially given by Kozen [?], is indeed complete. We write $\vdash_\mu \varphi$ if φ is provable in mu-calculus.

Proof system of mu-calculus extends propositional calculus with the following:

$$\text{(K)}\quad [a](\varphi \to \psi) \to ([a]\varphi \to [a]\psi) \qquad (\mathrm{Mu}_1)\quad \varphi[(\mu x.\varphi)/x] \to \mu x.\varphi \qquad (\mathrm{Mu}_2)\quad \frac{\varphi[\psi/x] \to \psi}{\mu x.\varphi \to \psi}$$

The way mu-calculus is defined in matching logic is similar as other modal logics. The theory $\mathsf{Mu} = (\Sigma, H)$ is a single-sorted theory and contains all definitions needed for the fixpoint constructs μ and ν. The variable set is Var. The signature set Σ contains

- a unary symbol a for every label $a \in$ Label;
- a constant symbol p for every atomic proposition $p \in$ AP;
- a constant symbol x for every variable $x \in$ Var.

Notice that we have both the variable x and the symbol x in matching logic. This is because in mu-calculus, a variable is either a standalone formula or the binding variable of a fixpoint construct. Mu-calculus assignments assign variables to sets, while matching logic valuations map variables to values (singleton sets). Therefore, we use the matching logic symbol x if it is a standalone formula

in mu-calculus, and use the variable x if it is a binding variable in $\mu x.\varphi$ or $\nu x.\varphi$. In addition, we define $\langle a\rangle\varphi \equiv a(\varphi)$ and $[a]\varphi \equiv \neg a(\neg\varphi)$. With the above definitions and notations,

Any mu-calculus formula φ is a closed matching logic pattern in theory Mu.

There is a one-to-one correspondence between mu-calculus structures and the intended models of theory Mu, as summarized in the following. Since all mu-calculus formulas are closed, it does not matter what matching logic valuation we use.

Mu-calculus	Matching logic theory Mu with intended semantics
state set S	carrier set S
binary relation $R_a \subseteq S \times S$	interpretation of a such that $s \in a_{\mathcal{M}}(t)$ if and only if sR_at
valuation $V\colon \mathrm{AP} \to 2^S$	interpretation of p such that $p_{\mathcal{M}} = V(p)$
assignment $g\colon \mathrm{Var} \to 2^S$	interpretation of x such that $x_{\mathcal{M}} = g(x)$
least fixpoint $[\![\mu x.\varphi]\!]_{\mathcal{M},g}$	intended interpretation $\bar{\rho}(\mu x.\varphi) = \mu[\![\varphi]\!]_{\mathcal{M},\rho}$ where ρ is any valuation (it makes no difference which ρ we use)
$s \in [\![\varphi]\!]_{\mathcal{M},g}$	$s \in \bar{\rho}(\varphi)$

Like hybrid modal logic (see Section 4.1), mu-calculus and the theory Mu admit the conservative extension result. Unlike hybrid modal logic, the proof involves both modal-theoretic and proof-theoretic approaches. The reason is that the above one-to-one correspondence only works for the *intended models*, not *all models*, of theory Mu. Therefore, we can conclude only that $\mathrm{Mu} \vDash \varphi$ implies $\vDash_\mu \varphi$, but not the other direction, as there may exist an unintended model, say $\mathcal{M}'$, which satisfies Mu and fails φ. Such unintended models $\mathcal{M}'$ are not considered at all in mu-calculus.

A proof-theoretic approach fills the gap. Careful readers may already notice that all axioms and rules in mu-calculus are provable in matching logic. Axiom (K) is provable as shown in Section 4.1. Axiom (Mu_1) is proved using (Fix_μ), and rule (Mu_2) is proved using (Lfp). Therefore, we conclude that $\vdash_\mu \varphi$ implies $\mathrm{Mu} \vdash \varphi$, because we can mimic every mu-calculus proofs in the matching logic theory Mu. The conservative extension for mu-calculus is then obtained by completeness of both mu-calculus and matching logic.

We point out that the above reasoning, as illustrated in Figure 3, is very general. In the following subsections, we will use the same technique to prove the conservative extension results for LTL, CTL, CTL*, etc, so it is important to understand what is needed for the proof. As shown in Figure 3, the diagram involves 7 steps, with 4 of them are either proved by definition, or established results about matching logic. Among the rest 3 steps, Step ($\Longrightarrow_2$) requires to show all axioms and proof rules of mu-calculus are provable in theory Mu, which is not trivial and can involve some intelligence. Step ($\Longrightarrow_6$) requires to show all mu-calculus models can be regarded as matching logic models (often the intended models) of theory Mu; the proof is often by carrying out structural induction on formulas, and thus we often need to prove both directions. Notice that there is a game we can play when defining theory Mu. We can add more axioms to Mu, which makes Step ($\Longrightarrow_2$) easier to prove but Step ($\Longrightarrow_6$) harder, as it rules out more models. If we do not have enough axioms, Step ($\Longrightarrow_2$) may not be provable. Finally, we rely on the completeness of mu-calculus (Step ($\Longrightarrow_1$)), which is also an established result but far from trivial.

Theorem 4.4 (Conservative extension for mu-calculus). *Let φ be a mu-calculus formula, and* Mu *be the matching logic theory for mu-calculus defined as above. Then, $\vdash_\mu \varphi$ if and only if* $\mathrm{Mu} \vdash \varphi$.

4.7 Linear temporal logic

Linear temporal logic (LTL) is parametric on a set AP of atomic propositions. In literature, the term LTL often refers to the *infinite-trace LTL*, where LTL formulas are interpreted on infinite traces. In program verification and especially runtime verification [?], finite execution traces play

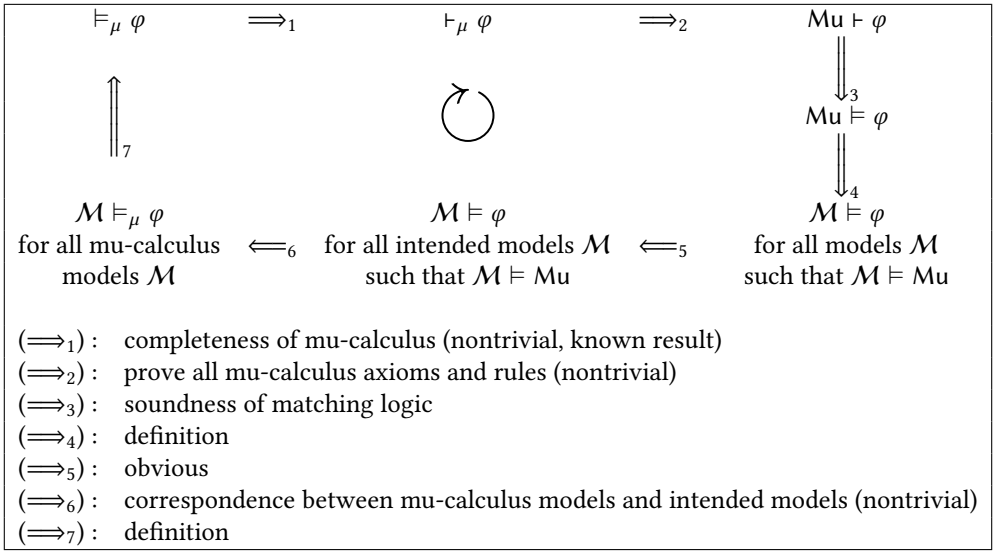

Fig. 3. A general method to prove conservative extension, using mu-calculus as an example.

an important role, and thus *finite-trace LTL* is considered. In this section, we will show how both finite- and infinite-trace LTL can be defined in a uniform way in matching logic.

4.7.1 Infinite-trace linear temporal logic. Readers should be more familiar with infinite-trace LTL, so let us consider that first. The syntax of infinite-trace LTL extends the syntax of propositional calculus with a "next" modality $\bigcirc$ and a "strong until" modality U_s:

$$\varphi ::= p \in \mathrm{AP} \mid \varphi \wedge \varphi \mid \neg\varphi \mid \bigcirc\varphi \mid \varphi \,\mathsf{U}_s\, \varphi$$

As usual, we define $\Diamond\varphi \equiv \mathit{true}\,\mathsf{U}_s\,\varphi$ and $\Box\varphi \equiv \neg(\Diamond\neg\varphi)$. Infinite-trace LTL formulas are interpreted on *infinite traces of sets of atomic propositions*, denoted as $\textsc{Traces}^\omega = [\mathbb{N} \to 2^{\mathrm{AP}}]$. We use $\alpha = \alpha_0\alpha_1\ldots$ to denote an infinite trace, and use the conventional notation $\alpha_{\geq i}$ to denote the suffix trace $\alpha_i\alpha_{i+1}\ldots$. Infinite-trace LTL semantics $\alpha \vDash_{\mathrm{infLTL}} \varphi$ is defined inductively as follows:

- $\alpha \vDash_{\mathrm{infLTL}} p$ if $p \in \alpha_0$ for atomic proposition p;
- $\alpha \vDash_{\mathrm{infLTL}} \varphi_1 \wedge \varphi_2$ if $\alpha \vDash_{\mathrm{infLTL}} \varphi_1$ and $\alpha \vDash \varphi_2$;
- $\alpha \vDash_{\mathrm{infLTL}} \neg\varphi$ if $\alpha \nvDash_{\mathrm{infLTL}} \varphi$;
- $\alpha \vDash_{\mathrm{infLTL}} \bigcirc\varphi$ if $\alpha_{\geq 1} \vDash_{\mathrm{infLTL}} \varphi$;
- $\alpha \vDash_{\mathrm{infLTL}} \varphi_1 \,\mathsf{U}_s\, \varphi_2$ if there is $j \geq 0$ such that $\alpha_{\geq j} \vDash_{\mathrm{infLTL}} \varphi_2$ and for every $i < j$, $\alpha_{\geq i} \vDash_{\mathrm{infLTL}} \varphi_1$.

An infinite-trace LTL formula φ is valid, denoted as $\vDash_{\mathrm{infLTL}} \varphi$, if $\alpha \vDash_{\mathrm{infLTL}} \varphi$ for every $\alpha \in \textsc{Traces}^\omega$. A sound and complete proof system of infinite-trace LTL is given as follows. We write $\vdash_{\mathrm{infLTL}} \varphi$ if an infinite-trace LTL formula φ is provable.

Proof system of infinite-trace LTL extends propositional calculus with the following:

$(\mathrm{K}_\bigcirc)$	$\bigcirc(\varphi_1 \to \varphi_2) \to (\bigcirc\varphi_1 \to \bigcirc\varphi_2)$	$(\mathrm{N}_\bigcirc)$	$\dfrac{\varphi}{\bigcirc\varphi}$
$(\mathrm{K}_\Box)$	$\Box(\varphi_1 \to \varphi_2) \to (\Box\varphi_1 \to \Box\varphi_2)$	$(\mathrm{N}_\Box)$	$\dfrac{\varphi}{\Box\varphi}$
$(\textsc{Fun})$	$\bigcirc\varphi \leftrightarrow \neg(\bigcirc\neg\varphi)$	(U_1)	$(\varphi_1 \,\mathsf{U}_s\, \varphi_2) \to \Diamond\varphi_2$
(U_2)	$(\varphi_1 \,\mathsf{U}_s\, \varphi_2) \leftrightarrow (\varphi_2 \vee (\varphi_1 \wedge \bigcirc(\varphi_1 \,\mathsf{U}_s\, \varphi_2)))$	$(\textsc{Ind})$	$\Box(\varphi \to \bigcirc\varphi) \to (\varphi \to \Box\varphi)$

We can define a single-sorted matching logic theory $\mathsf{infLTL} = (\Sigma, H)$ that faithfully captures infinite-trace LTL. The theory infLTL contains all definitions that are needed for the fixpoint constructs μ and ν. The signature Σ contains

- a unary symbol $\bullet$ called "strong-next"; We write $\bullet\varphi$ instead of $\bullet(\varphi)$;
- a constant symbol p for every atomic proposition $p \in \mathrm{AP}$.

We define the following derived matching logic constructs:

$$\circ\varphi \equiv \neg(\bullet\neg\varphi) \quad \Box\varphi \equiv \nu f.(\varphi \wedge \circ f) \quad \Diamond\varphi \equiv \mu f.(\varphi \vee \bullet f) \quad \varphi_1 \,\mathsf{U}_s\, \varphi_2 \equiv \mu f.(\varphi_2 \vee (\varphi_1 \wedge \bullet f))$$

In addition to axioms about fixpoint constructs, the axiom set H contains two more axioms

$$(\textsc{Lin}) \quad \bullet\varphi \rightarrow \circ\varphi \qquad\qquad (\textsc{Inf}) \quad \bullet\top$$

As we have seen in Section 4.5, axiom (Lin) and (Inf) give us linear and infinite traces, respectively. Notice that $\Box\varphi$ is not defined as a matching logic symbol, but defined using the fixpoint construct ν. By simple matching logic reasoning, we can show that $\Box\varphi = \neg\Diamond\neg\varphi$ and $\Diamond\varphi = \top \,\mathsf{U}_s\, \varphi$, as expected. Thanks to the above definitions and notations,

Any infinite-trace linear temporal logic formula is a pattern in theory infLTL.

As in Section 4.6, we show that $\vdash_{\mathrm{infLTL}} \varphi$ implies $\mathsf{infLTL} \vdash \varphi$ using the proof-theoretic approach, and that $\mathsf{infLTL} \vDash \varphi$ implies $\vDash_{\mathrm{infLTL}} \varphi$ using the model-theoretic approach, and let the completeness of both logics do the rest. Here, we only show the model-theoretic part. We define a matching logic model of theory infLTL called the *standard model*, denoted as $\mathcal{M}$, whose carrier set is the set $\textsc{Traces}^\omega$ of all infinite traces. It adopts the intended interpretation for fixpoint constructs. Other symbols and derived constructs in infLTL are interpreted as follows:

- $p_{\mathcal{M}} = \{\alpha \mid p \in \alpha_0\}$ for every atomic proposition $p \in \mathrm{AP}$;
- $\alpha \in \bullet_{\mathcal{M}}(\beta)$ if $\beta = \alpha_{\geq 1}$, i.e., β is the immediate surfix of α;

Notice that the standard model $\mathcal{M}$ satisfies (Lin) and (Fin), and thus is indeed a model of theory infLTL. Let ρ be any matching logic valuation. By structural induction on infinite-trace LTL formulas, one can show that for any infinite-trace LTL formula φ and an infinite trace α, $\alpha \vDash_{\mathrm{infLTL}} \varphi$ if and only if $\alpha \in \bar{\rho}(\varphi)$. And thus $\vDash_{\mathrm{infLTL}} \varphi$ if and only if $\mathcal{M} \vDash \varphi$. This finishes the reason diagram in Figure 3, and we conclude the conservative extension result for infinite-trace LTL.

Theorem 4.5 (Conservative Extension for Infinite-trace Linear Temporal Logic). *Let φ be an infinite-trace LTL formula and* infLTL *be the matching logic theory for infinite-trace LTL. Then, $\vdash_{\mathrm{infLTL}} \varphi$ if and only if* $\mathsf{infLTL} \vdash \varphi$.

4.7.2 Finite-trace linear temporal logic. Like infinite-trace LTL, finite-trace LTL formulas are interpreted on linear structures, i.e., traces. Unlike infinite-trace LTL, finite-trace LTL formulas are interpreted on finite traces. The syntax of finite-trace LTL is defined as follows:

$$\varphi ::= p \in \mathrm{AP} \mid \varphi \wedge \varphi \mid \neg\varphi \mid \circ\varphi \mid \varphi \,\mathsf{U}_w\, \varphi$$

As usual, define $\bullet\varphi \equiv \neg\circ\neg\varphi$, $\Box\varphi \equiv \varphi \,\mathsf{U}_w\, \mathit{false}$, and $\Diamond\varphi \equiv \neg(\Box\neg\varphi)$. Notice that we work with "weak until" U_w, different from infinite-trace LTL. As a result, in finite-trace LTL "always" $\Box$ is firstly defined, followed by "eventually" $\Diamond$. The two until's are different in that $\varphi_1 \,\mathsf{U}_s\, \varphi_2$ requires φ_2 eventually holds while $\varphi_1 \,\mathsf{U}_w\, \varphi_2$ does not. We refer readers to [?] for a more detailed discussion about finite-trace LTL and its relation with infinite-trace LTL.

Finite-trace LTL formulas are interpreted on nonempty finite traces of sets of atomic propositions, denoted as $\alpha = \alpha_0 \ldots \alpha_n$. We write $\textsc{Traces}^*$ to denote the set of all finite traces. The semantics $\alpha \vDash_{\mathrm{finLTL}} \varphi$ is defined similar to infinite-trace LTL. Notice $\circ\varphi$ holds in any singleton traces.

- $\alpha_0 \ldots \alpha_n \vDash_{\mathrm{finLTL}} p$ if $p \in \alpha_0$ for atomic proposition p;

- $\alpha_0 \dots \alpha_n \vDash_{\text{finLTL}} \varphi_1 \wedge \varphi_2$ if $\alpha_0 \dots \alpha_n \vDash_{\text{finLTL}} \varphi_1$ and $\alpha_0 \dots \alpha_n \vDash \varphi_2$;
- $\alpha_0 \dots \alpha_n \vDash_{\text{finLTL}} \neg\varphi$ if $\alpha_0 \dots \alpha_n \nvDash_{\text{finLTL}} \varphi$;
- $\alpha_0 \dots \alpha_n \vDash_{\text{finLTL}} \bigcirc\varphi$ if $n = 0$ or $\alpha_1 \dots \alpha_n \vDash_{\text{finLTL}} \varphi$;
- $\alpha_0 \dots \alpha_n \vDash_{\text{finLTL}} \varphi_1 \mathsf{U}_w \varphi_2$ if either for every $i \leq n$, $s_i \dots s_n \vDash_{\text{finLTL}} \varphi_1$, or there is $j \leq n$ such that $\alpha_j \dots \alpha_n \vDash_{\text{finLTL}} \varphi_2$ and for every $i < j$, $\alpha_i \dots \alpha_n \vDash_{\text{finLTL}} \varphi_1$.

Finite-trace LTL has a sound and complete proof system.

Proof system of finite-trace LTL extends propositional calculus with the following:

($\mathrm{K}_\bigcirc$)	$\bigcirc(\varphi_1 \to \varphi_2) \to (\bigcirc\varphi_1 \to \bigcirc\varphi_2)$	($\mathrm{N}_\bigcirc$)	$\dfrac{\varphi}{\bigcirc\varphi}$
($\mathrm{K}_\Box$)	$\Box(\varphi_1 \to \varphi_2) \to (\Box\varphi_1 \to \Box\varphi_2)$	($\mathrm{N}_\Box$)	$\dfrac{\varphi}{\Box\varphi}$
($\neg\bigcirc$)	$\neg\bigcirc\varphi \to \bigcirc\neg\varphi$	(coInd)	$\dfrac{\bigcirc\varphi \to \varphi}{\varphi}$
(Fix)	$(\varphi_1 \mathsf{U}_w \varphi_2) \leftrightarrow (\varphi_2 \vee (\varphi_1 \wedge \bigcirc(\varphi_1 \mathsf{U}_w \varphi_2)))$		

A matching logic theory $\mathsf{finLTL} = (\Sigma, H)$ that captures finite-trace LTL can be defined similarly as theory infLTL, while instead of axiom (Inf), we add axiom (Fin) to capture the finite-trace semantics. A conservative extension result is proved in the same way; the standard model of theory finLTL has Traces^ω as its carrier set.

We point out that the defining proof rule (coInd) in finite-trace LTL is provable from (Fin). In fact, we have $\vdash \lfloor \bigcirc\varphi \to \varphi \rfloor \to (\mu f . \bigcirc f \to \varphi)$ by axiom (Lfp), and the rest is by simple matching logic reasoning.

We end this subsection by stating the conservative extension result for finite-trace LTL.

Theorem 4.6 (Conservative Extension for Finite-trace Linear Temporal Logic). *Let φ be a finite-trace LTL formula and* finLTL *be the matching logic theory for finite-trace LTL defined as above. Then, $\vdash_{\text{finLTL}} \varphi$ if and only if* finLTL $\vdash \varphi$.

4.7.3 Unifying infinite- and finite-trace linear temporal logics. We propose a matching logic theory $\mathsf{LTL} = (\Sigma, H)$ that unifies both infinite- and finite-trace LTLs. The signature Σ contains a unary symbol $\bullet$ for "strong next", and conventional temporal modalities are defined in their usual way. The axiom set H contains (Lin) to capture the "linear trace" semantics of both LTLs, and one can always add axioms, (Inf) or (Fin), to obtain infinite- or finite-trace LTL. Therefore, the matching logic theory LTL provides a unified, flexible, and extensible way to reason about properties about linear structures. Instead of designing new logics for every subclasses linear structures of interest, we can use a fixed logic (the matching logic), and write axioms to restrict models and structures. We will see more examples in the following subsections.

4.8 Computation tree logic

Computation tree logic (CTL) is another popular logic to reason about properties of transition systems. Unlike LTL, CTL is a *branching time* logic. It interprets its formulas on infinite trees and has modalities that can quantify paths in a tree. The syntax of CTL is parametric on a set AP of atomic propositions and is defined as follows.

$$\varphi ::= p \in \mathrm{AP} \mid \varphi \wedge \varphi \mid \neg\varphi \mid \mathsf{AX}\varphi \mid \mathsf{EX}\varphi \mid \varphi\ \mathsf{AU}\ \varphi \mid \varphi\ \mathsf{EU}\ \varphi$$

Other CTL modalities are defined in the usual way:

$$\mathsf{EF}\varphi \equiv \mathit{true}\ \mathsf{EU}\ \varphi \qquad \mathsf{AG}\varphi \equiv \neg\mathsf{EF}\neg\varphi \qquad \mathsf{AF}\varphi \equiv \mathit{true}\ \mathsf{AU}\ \varphi \qquad \mathsf{EG}\varphi \equiv \neg\mathsf{AG}\neg\varphi$$

Every CTL modality contains two letters; the first means either "all-path" A or "one-path" E, and the second means "next" X, "until" U, "always" G, or "eventually" F. Therefore, AX is "all-path next", and EU is "one-path until", etc.

Let $\textsc{Trees}^{\omega}$ be the set of all infinite trees over AP. An infinite tree τ has sets of AP as its nodes; it is *infinite* in the sense that it has no leaves and every node has children. We write $\text{root}(\tau)$ to denote the root of τ. We write $\tau \longrightarrow \tau'$ if τ' is an immediate subtree of τ. CTL semantics $\tau \models_{\text{CTL}} \varphi$ is defined inductively as follows.

- $\tau \models_{\text{CTL}} p$ if $p \in \text{root}(\tau)$ for atomic proposition $p \in \text{AP}$;
- $\tau \models_{\text{CTL}} \varphi_1 \wedge \varphi_2$ if $\tau \models_{\text{CTL}} \varphi_1$ and $\tau \models_{\text{CTL}} \varphi_2$;
- $\tau \models_{\text{CTL}} \neg\varphi$ if $\tau \not\models_{\text{CTL}} \varphi$;
- $\tau \models_{\text{CTL}} \mathsf{AX}\varphi$ if for all τ' such that $\tau \longrightarrow \tau'$, $\tau' \models_{\text{CTL}} \varphi$;
- $\tau \models_{\text{CTL}} \mathsf{EX}\varphi$ if there exists τ' such that $\tau \longrightarrow \tau'$ and $\tau' \models_{\text{CTL}} \varphi$;
- $\tau \models_{\text{CTL}} \varphi_1 \mathsf{AU} \varphi_2$ if for all $\tau_0, \tau_1, \ldots$ such that $\tau = \tau_0 \longrightarrow \tau_1 \longrightarrow \ldots$, there exists $i \geq 0$ such that $\tau_i \models_{\text{CTL}} \varphi_2$ and for all $j < i$, $\tau_j \models_{\text{CTL}} \varphi_1$;
- $\tau \models_{\text{CTL}} \varphi_1 \mathsf{EU} \varphi_2$ if there exists $\tau_0, \tau_1, \ldots$ such that $\tau = \tau_0 \longrightarrow \tau_1 \longrightarrow \ldots$, and there exists $i \geq 0$ such that $\tau_i \models_{\text{CTL}} \varphi_2$ and for all $j < i$, $\tau_j \models_{\text{CTL}} \varphi_1$;

We write $\models_{\text{CTL}} \varphi$ if $\tau \models_{\text{CTL}} \varphi$ for all τ. CTL admits a sound and complete proof system shown as follows. We write $\vdash_{\text{CTL}} \varphi$ if φ is provable in CTL.

Proof system of computational tree logic extends propositional calculus with the following:

(CTL_1)	$\mathsf{EX}(\varphi_1 \vee \varphi_2) \leftrightarrow \mathsf{EX}\varphi_1 \vee \mathsf{EX}\varphi_2$
(CTL_2)	$\mathsf{AX}\varphi \leftrightarrow \neg(\mathsf{EX}\neg\varphi)$
(CTL_3)	$\varphi_1 \mathsf{EU} \varphi_2 \leftrightarrow \varphi_2 \vee (\varphi_1 \wedge \mathsf{EX}(\varphi_1 \mathsf{EU} \varphi_2))$
(CTL_4)	$\varphi_1 \mathsf{AU} \varphi_2 \leftrightarrow \varphi_2 \vee (\varphi_1 \wedge \mathsf{AX}(\varphi_1 \mathsf{AU} \varphi_2))$
(CTL_5)	$\mathsf{EX}\mathit{true} \wedge \mathsf{AX}\mathit{true}$
(CTL_6)	$\mathsf{AG}(\varphi_3 \longrightarrow (\neg\varphi_2 \wedge \mathsf{EX}\varphi_3)) \longrightarrow (\varphi_3 \longrightarrow \neg(\varphi_1 \mathsf{AU} \varphi_2))$
(CTL_7)	$\mathsf{AG}(\varphi_3 \longrightarrow (\neg\varphi_2 \wedge (\varphi_1 \longrightarrow \mathsf{AX}\varphi_3))) \longrightarrow (\varphi_3 \longrightarrow \neg(\varphi_1 \mathsf{EU} \varphi_2))$
(CTL_8)	$\mathsf{AG}(\varphi_1 \longrightarrow \varphi_2) \longrightarrow (\mathsf{EX}\varphi_1 \longrightarrow \mathsf{EX}\varphi_2)$

We can define a matching logic theory $\mathsf{CTL} = (\Sigma, H)$ that faithfully captures CTL. The theory CTL contains all definitions that are needed for fixpoint constructs μ and ν and the unary symbol "strong next" $\bullet$. Define "weak next" $\circ\varphi \equiv \neg\bullet\neg\varphi$ as usual. In addition, the signature Σ contains a constant symbol p for every atomic proposition $p \in \text{AP}$. The axiom set H contains fixpoint axioms plus axiom $(\textsc{Inf})$ to capture the infinite tree semantics of CTL. In other words, remove axiom $(\textsc{Lin})$ from theory infLTL and we obtain the theory CTL. We define CTL modalities as derived constructs in matching logic as follows:

$$\mathsf{AX}\varphi \equiv \circ\varphi \qquad \mathsf{EX}\varphi \equiv \bullet\varphi \qquad \varphi_1 \mathsf{AU} \varphi_2 \equiv \mu f.\varphi_2 \vee (\varphi_1 \wedge \circ f) \qquad \varphi_1 \mathsf{EU} \varphi_2 \equiv \mu f.\varphi_2 \vee (\varphi_1 \wedge \bullet f)$$

With the above definitions and notations,

Any computational tree logic formula is a matching logic pattern of theory CTL.

The standard model of theory CTL has the set $\textsc{Trees}^{\omega}$ of all infinite trees as its carrier set. It adopts intended semantics for fixpoint constructs μ and ν. Other symbols are interpreted as follows:

- $p_{\mathcal{M}} = \{\tau \mid p \in \text{root}(\tau)\}$ for atomic proposition $p \in \text{AP}$;
- $\tau \in \bullet_{\mathcal{M}}(\tau')$ if $\tau \longrightarrow \tau'$;

One can prove that CTL validity coincides with the validity in this standard model. In addition, one can prove that all CTL proof rules and axioms are provable in matching logic. As in Section 4.7, this gives us the following conservative extension result for CTL.

Theorem 4.7 (Conservative Extension for Computational Tree Logic). *Let φ be a computational tree logic formula and* CTL *be the matching logic theory for computational tree logic defined as above. Then, $\vdash_{CTL} \varphi$ if and only if* CTL $\vdash \varphi$.

Before we end this subsection, we point out that matching logic provides a uniform way to study and play with variants of CTL. For example, CTL as presented here adopts infinite-tree semantics. One can consider a variant of CTL with finite-tree semantics, and it cannot be easier to do that in matching logic. One just needs to replace axiom (Inf) in theory CTL with axiom (Fin), or simply remove it to capture both finite- and infinite CTLs. Instead of designing a new logic, one writes axioms to capture the intended semantics, and matching logic offers a sound and complete deduction for free. Even though the the axiomatization may not completely capture the intended semantics, it can faithfully and completely capture logics or calculi with complete deduction that are specifically designed for that semantics.

4.9 Propositional dynamic logic

Propositional dynamic logic (PDL) is an extension of modal logic to reason about programs. Its syntax is parametric on a set AP of atomic propositions and a set APGM of atomic programs. PDL has two types of formulas; *propositional formulas* are similar to formulas in modal logic or mu-calculus, and *program formulas (terms)* represent programs built from atomic programs and primitive regular expression operators.

$$\begin{array}{ll} \textit{propositional formulas} & \varphi ::= p \in \mathrm{AP} \mid \varphi \rightarrow \varphi \mid \textit{false} \mid [\alpha]\varphi \\ \textit{program formulas} & \alpha ::= a \in \mathrm{APGM} \mid \alpha\,;\alpha \mid \alpha \cup \alpha \mid \alpha^* \mid \alpha? \end{array}$$

Common propositional connectives can be defined from $\varphi \rightarrow \varphi$ and *false* in the usual way. Common program constructs such as if-then-else, while-do, and repeat-until statements can also be defined using the four primitive constructs. These are not our focus, so we refer readers to PDL literatures such as [**?**] for details. Define $\langle a\rangle\varphi \equiv \neg[\alpha](\neg\varphi)$ as in mu-calculus.

PDL formulas are interpreted on Kripke frames $\mathcal{M} = (M, [\![\_]\!]_{\mathcal{M}})$ where M is a state set and $[\![\_]\!]_{\mathcal{M}}$ is a meaning function that

- maps every atomic proposition $p \in \mathrm{AP}$ to a set of states $[\![p]\!]_{\mathcal{M}} \subseteq M$;
- maps every atomic programs $a \in \mathrm{APGM}$ to a binary relation on states $[\![a]\!]_{\mathcal{M}} \subseteq M \times M$.

Then, the meaning function is extended to all propositional and program formulas in the following mutual inductive way:

- $[\![\varphi_1 \rightarrow \varphi_2]\!]_{\mathcal{M}} = (M \setminus [\![\varphi_1]\!]_{\mathcal{M}}) \cup [\![\varphi_2]\!]_{\mathcal{M}}$;
- $[\![\textit{false}]\!]_{\mathcal{M}} = \emptyset$;
- $[\![[\alpha]\varphi]\!]_{\mathcal{M}} = \{s \mid \text{for all } t \in M \text{ such that } (s,t) \in [\![\alpha]\!]_{\mathcal{M}},\ t \in [\![\varphi]\!]_{\mathcal{M}}\ \}$
- $[\![\alpha\,;\beta]\!]_{\mathcal{M}} = \{(s,t) \mid \text{ there exists a state } s' \text{ such that } (s,s') \in [\![\alpha]\!]_{\mathcal{M}} \text{ and } (s',t) \in [\![\beta]\!]_{\mathcal{M}}\ \}$
- $[\![\alpha \cup \beta]\!]_{\mathcal{M}} = [\![\alpha]\!]_{\mathcal{M}} \cup [\![\beta]\!]_{\mathcal{M}}$
- $[\![\alpha^*]\!]_{\mathcal{M}} = \bigcup_{n\geq 0} ([\![\alpha]\!]_{\mathcal{M}})^n$
- $[\![\varphi?]\!]_{\mathcal{M}} = [\![\varphi]\!]_{\mathcal{M}} \times [\![\varphi]\!]_{\mathcal{M}}$

A PDL formula φ is valid, denoted as $\vDash_{\mathrm{PDL}} \varphi$, if it holds in all Kripke frames. PDL has a sound and complete proof system. We write $\vdash_{\mathrm{PDL}} \varphi$ if φ is provable in PDL.

Proof system of propositional dynamic logic extends propositional calculus with the following:

(PDL_1)	$[\alpha](\varphi_1 \rightarrow \varphi_2) \rightarrow ([\alpha]\varphi_1 \rightarrow [\alpha]\varphi_2)$	(PDL_2)	$[\alpha](\varphi_1 \wedge \varphi_2) \leftrightarrow ([\alpha]\varphi_1 \wedge [\alpha]\varphi_2)$
(PDL_3)	$[\alpha \cup \beta]\varphi \leftrightarrow [\alpha]\varphi \wedge [\beta]\varphi$	(PDL_4)	$[\alpha\,;\beta]\varphi \leftrightarrow [\alpha][\beta]\varphi$
(PDL_5)	$[\psi?]\varphi \leftrightarrow (\psi \rightarrow \varphi)$	(PDL_6)	$\varphi \wedge [\alpha][\alpha^*]\varphi \leftrightarrow [\alpha^*]\varphi$
(PDL_7)	$\varphi \wedge [\alpha^*](\varphi \rightarrow [\alpha]\varphi) \rightarrow [\alpha^*]\varphi$	(Gen)	$\dfrac{\varphi}{[\alpha]\varphi}$

We compare PDL formulas $[\alpha]\varphi$ with mu-calculus formulas $[a]\varphi$. In mu-calculus, the action set is a discrete set and actions have no structure; while in PDL, programs have structures and are constructed from atomic ones with regular expression operators. Recall that in Section 4.6, we define a unary symbol a for every mu-calculus action a and define $\langle a\rangle\varphi \equiv a(\varphi)$. This is known as a *shallow embedding*. In PDL, we adopt a different approach called *deep embedding*, which we elaborate in detail as follows.

We can define a matching logic theory $\mathrm{PDL} = (\mathbb{\Sigma}, H)$ that faithfully captures PDL. The signature $\mathbb{\Sigma} = (S, \Sigma)$ contains a sort set $S = \{state, pgm\}$ with a sort *state* for propositional formulas and a sort *pgm* for program formulas. The symbol set Σ contains

- a constant symbol $p \in \Sigma_{\lambda,\, state}$ for atomic proposition $p \in \mathrm{AP}$;
- a constant symbol $a \in \Sigma_{\lambda,\, pgm}$ for atomic program $a \in \mathrm{AP}_{\mathrm{PGM}}$;
- a unary symbol $\bullet \in \Sigma_{state,\, state}$ called "strong next";
- a binary symbol $\_\,;\,\_ \in \Sigma_{pgm\ pgm,\, pgm}$;
- a binary symbol $\_ \cup \_ \in \Sigma_{pgm\ pgm,\, pgm}$;
- a unary symbol $\_^* \in \Sigma_{pgm,\, pgm}$;
- a unary symbol $\_? \in \Sigma_{state,\, pgm}$;

In addition, the theory PDL contains all definitions needed for fixpoint constructs. Define the notions $\langle\alpha\rangle\varphi \equiv \bullet(\alpha, \varphi)$ and $[\alpha]\varphi \equiv \neg\langle\alpha\rangle(\neg\varphi)$. With the above definitions and notations,

> *Any propositional formula of PDL is a matching logic pattern of sort state;*
> *Any program formula of PDL is a matching logic pattern of sort pgm;*

Theory PDL is called a deep embedding because it defines the concrete syntax of PDL programs in sort *pgm*, and has separate axioms defining their semantics. The axiom set H contains the following four defining axioms, one for each PDL program construct.

$$\begin{array}{llll} (\textsc{Choice}) & [\alpha \cup \beta]\varphi = [\alpha]\varphi \wedge [\beta]\varphi & (\textsc{Seq}) & [\alpha\,;\,\beta]\varphi = [\alpha][\beta]\varphi \\ (\textsc{Test}) & [\psi?]\varphi = (\psi \rightarrow \varphi) & (\textsc{Iter}) & [\alpha^*]\varphi = \nu f.(\varphi \wedge [\alpha]f) \end{array}$$

Obviously, axioms (Choice), (Seq), and (Test) imply PDL axioms (PDL_3), (PDL_4), and (PDL_5), respectively. By easy fixpoint reasoning, one can prove that axiom (Iter) implies PDL axioms (PDL_6) and (PDL_7). In addition, (PDL_1), (PDL_2), and (Gen) are general properties about "strong next" $\bullet$ and also provable in theory PDL. Therefore, $\vdash_{\mathrm{PDL}} \varphi$ implies $\mathrm{PDL} \vdash \varphi$.

We claim that the four matching logic axioms for PDL program constructs defined in the above are more natural to understand and easier to design than the original PDL axioms. The PDL proof system is a blend of general modal logic reasoning, e.g., (PDL_1), (PDL_2), and (Gen), and specific axioms about program constructs, e.g, (PDL_3) – (PDL_7). Besides, axioms (PDL_6) are (PDL_7) are more like properties rather than a definition, because the formula $[\alpha^*]\varphi$ has multiple occurrences, while axiom (Iter) is clearly a definition.

One may argue that (Iter) uses the gfp construct ν and relies on axioms (Fix) and (Gfp), and (PDL_6) and (PDL_7) share the same style. We agree. And that is exactly why we think one should work in *a uniform and fixed logic* where general axioms about fixpoints can be defined. Then, we can use these axioms to reason about fixpoint properties and develop automatic tools and provers, rather than designing new logics and tools which all have their own ways to deal with fixpoints or induction axioms. We showed how mu-calculus, LTL, CTL, and PDL can be completely captured in matching logic with fixpoint axioms, and we hope it demonstrate that matching logic can be considered as a candidate of such a uniform and fixed logic.

We end this subsection with a conservative extension result for PDL. The result can be shown in the same way as in mu-calculus (see Figure 3), and we omit the proof.

Theorem 4.8 (Conservative Extension for Propositional Dynamic Logic). *Let φ be a propositional formula in propositional dynamic logic and* PDL *is the matching logic for propositional dynamic logic defined as above. Then, $\vdash_{PDL} \varphi$ if and only if* PDL $\vdash \varphi$.

4.10 Reachability logic (2 pages)

Reachability logic is a language-independent proof system for deriving reachability properties of systems and programs [?]. Its defining feature is the (Circularity) proof rule that supports reasoning about circular behavior of iterative and recursive program constructs. In historical literature [?], reachability logic is proposed alongside matching logic for program verification, where matching logic is used for defining static structure and program configurations, while reachability logic is used for reasoning about dynamic behavior.

In this section, we make the first step towards unifying reachability logic and matching logic. As in previous subsections, we aim at a conservative extension result for reachability logic, using the same method shown in Figure 3. We define reachability logic as a matching logic theory and show matching logic models subsume all reachability logic models. What is harder, and what we postpone to future work, is showing that all provable reachability rules are also provable in matching logic. In this paper, we take as examples some reachability rules that are originated from program verification problems, and prove them in matching logic. Readers will get a flavor of how reachability proofs using (Circularity) rule can be carried out in matching logic with fixpoint axioms (Fix) and (Lfp). In the next, we present the syntax and semantics of unconditional reachability logic in the way that fits the best with matching logic.

Let $\mathbb{\Sigma}$ be a matching logic signature used to specify static program configurations. Depending on the target programming language of interest, the signature $\mathbb{\Sigma}$ may have multiple sorts and symbols, among which there is a distinguished sort *Cfg*. Let $\mathcal{M} = (M, \_{}_{\mathcal{M}})$ be a matching logic model of signature $\mathbb{\Sigma}$ called the *underlying configuration model*. In particular, the carrier set M_{Cfg} is the domain of all configurations of the target language. The syntax and semantics of reachability logic is parametric in the signature $\mathbb{\Sigma}$ and the underlying configuration model $\mathcal{M}$. An *unconditional reachability rule*, or simply a *rule*, has the form $\varphi_1 \Rightarrow \varphi_2$ where φ_1 and φ_2 are patterns of sort *Cfg*. A *reachability system*, denoted as T, is a set of rules; it then yields a transition system over configurations, denoted as $\mathcal{T} = (M_{Cfg}, \rightarrow)$, where M_{Cfg} is the domain of all configurations in the underlying configuration model. The transition relation $\rightarrow$ is defined such that $s \rightarrow t$ if and only if there exists a rule $\varphi_1 \Rightarrow \varphi_2$ in T and a matching logic valuation ρ such that $s \in \bar{\rho}(\varphi_1)$ and $t \in \bar{\rho}(\varphi_2)$. Let $\rightarrow^* = \bigcup_{k \geq 0} (\rightarrow)^k$ be the transitive and reflexive closure of $\rightarrow$.

Let $\psi_1 \Rightarrow \psi_2$ be any unconditional reachability rule. We say it is ρ-valid, denoted as $T, \rho \vDash_{\text{uRL}} \psi_1 \Rightarrow \psi_2$, if for every configuration s such that $s \in \bar{\rho}(\varphi_1)$, either there is an infinite transition sequence $s \rightarrow s' \rightarrow s'' \rightarrow \ldots$ in the transition system $\mathcal{T}$ yielded by T, or there exists a configuration t such that $s \rightarrow^* t$ and $t \in \bar{\rho}(\varphi_2)$. Rule $\psi_1 \Rightarrow \psi_2$ is valid, denoted as $T \vDash_{\text{uRL}} \psi_1 \Rightarrow \psi_2$, if it is ρ-valid for every valuation ρ. Notice that validity in reachability logic is defined in the spirit of partial correctness.

Unconditional reachability logic has a sound and relatively complete proof system as shown below. Notice the proof system derives more general sequents of the form $A \vdash_C \varphi_1 \Rightarrow \varphi_2$ where A and C are sets of rules. We call rules in A *axioms* and rules in C *circularities*.

Proof system of unconditional reachability logic is parametric in a matching logic signature $\mathbb{\Sigma}$ for configurations and an underlying configuration model $\mathcal{M}$; it contains the following rules:

(Axiom) $\dfrac{\cdot}{A \vdash_C \varphi_1 \wedge \psi \Rightarrow \varphi_2 \wedge \psi}$ if $(\varphi_1 \Rightarrow \varphi_2) \in A$ and ψ is a predicate pattern

(Reflexivity) $\dfrac{\cdot}{A \vdash_\emptyset \varphi \Rightarrow \varphi}$ (Transitivity) $\dfrac{A \vdash_C \varphi_1 \Rightarrow \varphi_2 \quad A \cup C \vdash_\emptyset \varphi_2 \Rightarrow \varphi_3}{A \vdash_C \varphi_1 \Rightarrow \varphi_3}$

(Consequence) $\dfrac{\mathcal{M} \vDash \varphi_1 \rightarrow \varphi_1' \quad A \vdash_C \varphi_1' \Rightarrow \varphi_2' \quad \mathcal{M} \vDash \varphi_2' \rightarrow \varphi_2}{A \vdash_C \varphi_1 \Rightarrow \varphi_2}$

(Abstraction) $\dfrac{A \vdash_C \varphi_1 \Rightarrow \varphi_2}{A \vdash_C (\exists x.\varphi_1) \Rightarrow \varphi_2}$ if $x \notin FV(\varphi_2)$

(Circularity) $\dfrac{A \vdash_{C \cup \{\varphi_1 \Rightarrow \varphi_2\}} \varphi_1 \Rightarrow \varphi_2}{A \vdash_C \varphi_1 \Rightarrow \varphi_2}$ (Case Analysis) $\dfrac{A \vdash_C \varphi_1 \Rightarrow \varphi \quad A \vdash_C \varphi_2 \Rightarrow \varphi}{A \vdash_C \varphi_1 \vee \varphi_2 \Rightarrow \varphi}$

Rule $\varphi_1 \Rightarrow \varphi_2$ is provable in the reachability system T, denoted as $T \vdash_{\text{uRL}} \varphi_1 \Rightarrow \varphi_2$, if the sequent $T \vdash_\emptyset \varphi_1 \Rightarrow \varphi_2$ can be derived in the above proof system.

Theorem 4.9 (Soundness and completeness of unconditional reachability logic). *Let $\mathbb{\Sigma}$ be a matching logic signature for configurations and $\mathcal{M}$ be the underlying configuration model. Let T be a reachability system and $\varphi_1 \Rightarrow \varphi_2$ be an unconditional reachability rule. Then $T \vDash_{uRL} \varphi_1 \Rightarrow \varphi_2$ if and only if $T \vdash_{uRL} \varphi_1 \Rightarrow \varphi_2$.*

The completeness of unconditional reachability logic is a *relative* one because the proof system consults the underlying configuration model $\mathcal{M}$ for validity in (Consequence) rule. In other words, unconditional reachability logic is complete relative to the completeness of $\mathcal{M}$. If $\mathcal{M}$ can be completely axiomatized by a recursively enumerable set of axioms in matching logic, then $\mathcal{M} \vDash \varphi \rightarrow \varphi'$ is decidable, and the unconditional reachability logic (parametric on $\mathcal{M}$) becomes "absolutely" complete, i.e., there exists an algorithm that decides the validity of reachability rules. In practice, however, $\mathcal{M} \vDash \varphi \rightarrow \varphi'$ is often undecidable, and thus no algorithm can decide the validity of reachability rules. What Theorem 4.9 tells us is that this incompleteness origins in the undecidability of the underlying configuration model $\mathcal{M}$, not reachability logic itself.

To capture reachability logic in matching logic, we extends the signature $\mathbb{\Sigma}$ with the unary "strong next" symbol $\bullet \in \Sigma_{Cfg,Cfg}$ as well as fixpoint constructs μ and ν. The "strong next" symbol $\bullet$ allows us to specify in matching logic not just static structural properties about configurations, but also reachability rules and dynamic behavior of transition systems. Denote this extended signature as $\mathbb{\Sigma}^{\rightarrow}$. As usual, define $\Diamond\varphi \equiv \mu f.(\varphi \vee \bullet f)$ the same as the CTL modality EF (see Section 4.8) that means "one-path eventually". Reachability rules are defined as a derived construct as follows:

$$\varphi_1 \Rightarrow \varphi_2 \quad \equiv \quad \varphi_1 \rightarrow ((\neg \mu f.\bigcirc f) \vee \Diamond \varphi_2)$$

Recall that the intended semantics of $\mu f.\bigcirc f$ is the set of states that *"all-path" terminate* (see Section 4.5), so $\neg \mu f.\bigcirc f$ is the set of states that *"one-path" diverge*. Thus the intuition of the above definition is that any state in φ_1 either "one-path" diverges, or "one-path" eventually reaches φ_2, which coincides with the reachability logic semantics. It is not hard to show that

$$\varphi_1 \Rightarrow \varphi_2 \quad \equiv \quad \varphi_1 \rightarrow ((\neg \mu f.\bigcirc f) \vee \Diamond \varphi_2) \quad = \quad \mu f.\bigcirc f \rightarrow (\varphi_1 \rightarrow \Diamond \varphi_2)$$

The rightmost form is convenient in fixpoint reasoning. With the above definitions and notations,

Any reachability logic rule $\varphi_1 \Rightarrow \varphi_2$ is a matching logic pattern of sort Cfg.

Let uRL be a matching logic theory of signature $\mathbb{\Sigma}^{\rightarrow}$. We use it to capture unconditional reachability logic. The theory uRL contains all definitions needed for fixpoint constructs. In addition, it contains all valid patterns in the underlying configuration model $\mathcal{M}$ as axioms. This makes all implications needed for (Consequence) rule provable in theory uRL.

Let T be a reachability system and $\mathcal{T} = (M_{Cfg}, \rightarrow)$ be its yielded transition system. It is not hard to phrase the transition system $\mathcal{T}$ as a matching logic model of theory uRL and prove that $\mathcal{T} \models \varphi_1 \Rightarrow \varphi_2$ if and only if $T \vDash_{\text{uRL}} \varphi_1 \Rightarrow \varphi_2$. Extend theory uRL by adding all reachability rules in T as axioms and denote the extended theory as uRL_T. It follows that $\mathcal{T} \models \text{uRL}_T$, and that $\text{uRL}_T \models \varphi_1 \Rightarrow \varphi_2$ implies $T \vDash_{\text{uRL}} \varphi_1 \Rightarrow \varphi_2$.

We conjecture the following conservative extension theorem for reachability logic, whose proof is postponed to future work.

Conjecture 4.10 (Conservative Extension for Unconditional Reachability Logic). *Let $\mathbb{\Sigma}$ be a matching logic signature of configurations, $\mathcal{M}$ be an underlying configuration model, $\varphi_1 \Rightarrow \varphi_2$ be a reachability rule, T be a reachability system, and uRL_T be the corresponding matching logic theory all as defined above. Then, $T \vdash_{uRL} \varphi_1 \Rightarrow \varphi_2$ if and only if $\text{uRL}_T \vdash \varphi_1 \Rightarrow \varphi_2$.*

To prove the conjecture, it suffices to prove that $T \vdash_{\text{uRL}} \varphi_1 \Rightarrow \varphi_2$ implies $\text{uRL}_T \vdash \varphi_1 \Rightarrow \varphi_2$. In the following, we use an example originated from program verification problems to show how reachability proofs, especially application of (Circularity) rule, can be carried out in matching logic.

(Xiaohong) Finish the detail of example SUM

The invariant rule we want to prove is that

$$\exists n.(\varphi(n) \wedge n \geq 0) \Rightarrow \psi \qquad \text{(Invariant)}$$

Firstly, let us list some facts that are needed in the proof. Let us assume that the following two patterns are proved in advance.

$$\varphi(0) \rightarrow \Diamond\psi \qquad \text{(Base Case)}$$

$$\varphi(n) \wedge n \geq 1 \rightarrow \bullet^k \varphi(n-1) \qquad \text{for some } k \geq 1 \qquad \text{(Loop Body)}$$

The following patterns are either valid in the underlying configuration model $\mathcal{M}$ or provable using fixpoint axioms.

$$\exists n(\varphi(n) \wedge n \geq 0) = \varphi(0) \vee \exists n.(\varphi(n) \wedge n \geq 1) \qquad \text{(Domain)}$$

$$\Diamond\psi_1 = \bullet^k \psi_1 \qquad \text{(Next Eventually)}$$

$$\forall n.\bigcirc^k \psi_1 = \bigcirc^k \forall n.\psi_1 \qquad \text{(Comm)}$$

$$\mu f.\bigcirc f = \mu f.\bigcirc^k f \qquad \text{(Fix Next)}$$

$$\bigcirc^k(\psi_1 \rightarrow \psi_2) \wedge \bullet^k \psi_1 \rightarrow \bullet^k \psi_2 \qquad \text{(Progress)}$$

Finally we show the proof of (Invariant) in Figure 4. The symbol "$\Longleftarrow$" in the proof should be read as "to prove the above, it suffices to prove the below".

5 CONTEXTS AND SEMANTICS OF $\mathbb{K}$ (4 PAGES)

5.1 $\mathbb{K}$ overview

$\mathbb{K}$ is a rewrite-based executable semantics framework for programming language design. We use the language IMP in Figure 5 as our running example (with minor modification on its syntax) to illustrate how to define programming languages and verify programs in $\mathbb{K}$.

A complete $\mathbb{K}$ definition for IMP is shown in Figure 6, consisting of two $\mathbb{K}$ modules `IMP-SYNTAX` and `IMP`. The module `IMP-SYNTAX` defines the syntax of the language using the conventional BNF grammar, where terminals are in quotes. Syntax productions are separated by the "`|`" and "`>`", where "`|`" means the two productions have the same precedence while "`>`" means the previous

$$\exists n.(\varphi(n) \wedge n \geq 0) \Rightarrow \psi$$

$\Longleftarrow$ by definition

$$\exists n.(\varphi(n) \wedge n \geq 0) \rightarrow (\mu f. \bigcirc f \rightarrow \Diamond\psi)$$

$\Longleftarrow$ by (Domain)

$$\varphi(0) \vee \exists n.(\varphi(n) \wedge n \geq 1) \rightarrow (\mu f. \bigcirc f \rightarrow \Diamond\psi)$$

$\Longleftarrow$ by propositional reasoning

$$(\varphi(0) \rightarrow (\mu f. \bigcirc f \rightarrow \Diamond\psi)) \wedge (\exists n.(\varphi(n) \wedge n \geq 1) \rightarrow (\mu f. \bigcirc f \rightarrow \Diamond\psi))$$

$\Longleftarrow$ by (Base Case)

$$\exists n.(\varphi(n) \wedge n \geq 1) \rightarrow (\mu f. \bigcirc f \rightarrow \Diamond\psi)$$

$\Longleftarrow$ by FOL reasoning

$$\mu f. \bigcirc f \rightarrow \forall n.(\varphi(n) \wedge n \geq 1 \rightarrow \Diamond\psi)$$

$\Longleftarrow$ by (Fix Next)

$$\mu f. \bigcirc^k f \rightarrow \forall n.(\varphi(n) \wedge n \geq 1 \rightarrow \Diamond\psi)$$

$\Longleftarrow$ by (Lfp)

$$\bigcirc^k \forall n.(\varphi(n) \wedge n \geq 1 \rightarrow \Diamond\psi) \rightarrow \forall n.(\varphi(n) \wedge n \geq 1 \rightarrow \Diamond\psi)$$

$\Longleftarrow$ by (Comm)

$$\forall n. \bigcirc^k (\varphi(n) \wedge n \geq 1 \rightarrow \Diamond\psi) \rightarrow \forall n.(\varphi(n) \wedge n \geq 1 \rightarrow \Diamond\psi)$$

$\Longleftarrow$ by FOL reasoning

$$\bigcirc^k (\varphi(n-1) \wedge n - 1 \geq 1 \rightarrow \Diamond\psi) \rightarrow (\varphi(n) \wedge n \geq 1) \rightarrow \Diamond\psi$$

$\Longleftarrow$ by propositional reasoning

$$\bigcirc^k (\varphi(n-1) \wedge n - 1 \geq 1 \rightarrow \Diamond\psi) \wedge \varphi(n) \wedge n \geq 1 \rightarrow \Diamond\psi$$

$\Longleftarrow$ by (Loop Body)

$$\bigcirc^k (\varphi(n-1) \wedge n - 1 \geq 1 \rightarrow \Diamond\psi) \wedge \bullet^k \varphi(n-1) \wedge n \geq 1 \rightarrow \Diamond\psi$$

$\Longleftarrow$ by propositional reasoning

$$\bigcirc^k (\varphi(n-1) \wedge n \geq 2 \rightarrow \Diamond\psi) \wedge \bullet^k \varphi(n-1) \wedge n \geq 2 \rightarrow \Diamond\psi$$
$$\wedge \quad \bigcirc^k (\varphi(n-1) \wedge n \geq 2 \rightarrow \Diamond\psi) \wedge \bullet^k \varphi(n-1) \wedge n = 1 \rightarrow \Diamond\psi$$

$\Longleftarrow$ by propositional reasoning

$$\bigcirc^k (\varphi(n-1) \wedge n \geq 2 \rightarrow \Diamond\psi) \wedge \bullet^k \varphi(n-1) \wedge n \geq 2 \rightarrow \Diamond\psi$$
$$\wedge \quad \bullet^k \varphi(0) \rightarrow \Diamond\psi$$

$\Longleftarrow$ by (Next Eventually)

$$\bigcirc^k (\varphi(n-1) \wedge n \geq 2 \rightarrow \Diamond\psi) \wedge \bullet^k \varphi(n-1) \wedge n \geq 2 \rightarrow \Diamond\psi$$
$$\wedge \quad \bullet^k \varphi(0) \rightarrow \bullet^k \Diamond\psi$$

$\Longleftarrow$ by (Base Case) and frame reasoning

$$\bigcirc^k (\varphi(n-1) \wedge n \geq 2 \rightarrow \Diamond\psi) \wedge \bullet^k \varphi(n-1) \wedge n \geq 2 \rightarrow \Diamond\psi$$

$\Longleftarrow$ by matching logic reasoning

$$\bigcirc^k (\varphi(n-1) \wedge n \geq 2 \rightarrow \Diamond\psi) \wedge \bullet^k (\varphi(n-1) \wedge n \geq 2) \rightarrow \Diamond\psi$$

$\Longleftarrow$ by (Progress)

$$\bullet^k (\Diamond\psi) \rightarrow \Diamond\psi$$

$\Longleftarrow$ by (Next Eventually)

QED

Fig. 4. A proof of (Invariant).

$$\begin{array}{lll} \mathit{Exp} & ::= & \mathit{Id} \mid \mathit{Int} \mid \mathit{Exp} + \mathit{Exp} \mid \mathit{Exp} - \mathit{Exp} \\ \mathit{Stmt} & ::= & \mathit{Id} = \mathit{Exp}; \mid \mathit{StmtStmt} \mid \{\ \mathit{Stmt}\ \} \mid \{\} \mid \texttt{if}\ (\ \mathit{Exp}\ )\ \mathit{StmtStmt} \mid \texttt{while}\ (\ \mathit{Exp}\ )\ \mathit{Stmt} \end{array}$$

Fig. 5. The syntax of a simple imperative language IMP.

production has higher precedence (binds tighter) than the one that follows. In other words, in the language IMP, all language constructs bind tighter than the sequential operator. `Int` and `Id`

are two built-in categories of integers and identifiers (program variables), respectively. `Exp` is the category of expressions, which subsumes `Int` and `Id`, and contains two other productions for plus and minus. `Pgm` is the category of IMP programs. A wellformed IMP program declares a list of program variables in the beginning, followed by a statement. `Ids` is the category for lists of program variables, and it is defined using $\mathbb{K}$'s built-in template `List`. The first argument is the base category `Id`, and second argument is the separating character `","`.

Attributes are wrapped with braces "`[`" and "`]`". Some attributes are only for parsing purpose while others may carry additional semantic meaning and affect how $\mathbb{K}$ executes programs. The attribute `left` means that "`+`" and "`-`" are left-associative, so `1 - 2 + 3` should be parsed as `(1 - 2) + 3`. The attribute `strict` defines evaluation contexts. When $\mathbb{K}$ sees the expression e_1 `+` e_2 (and similarly e_1 `-` e_2), it first evaluates e_1 to an integer i_1 and e_2 to an integers i_2 in a *fully nondeterministic* way, and then evaluates i_1 + i_2. For example, there are in total `3! = 6` different orders to evaluate the expression `((1 + 2) + (3 + 4)) + (5 + 6)`, because the most inner three parentheses must be evaluated first, and they can be evaluated in any order. The attribute `strict(1)` defines evaluation contexts only for the first argument. Therefore, when $\mathbb{K}$ sees an if-statement `if(`b`)` P Q, it only evaluates the condition b and keeps the branches P and Q untouched. In other words, the two branches of if-statements are *frozen* and will not be evaluated until the condition becomes a value. The attribute `bracket` tells $\mathbb{K}$ that certain productions are only used for grouping, and $\mathbb{K}$ will not generate nodes in its internal abstract syntax trees for those productions. Here, parentheses "`( )`" are used to group arithmetics expressions while curly brackets "`{ }`" are used to group program statements. The empty curly bracket "`{}`" represents the empty statement.

```
module IMP-SYNTAX
imports DOMAINS-SYNTAX
syntax Exp  ::= Int | Id
| Exp "+" Exp             [left, strict]
| Exp "-" Exp             [left, strict]
| "(" Exp ")"                  [bracket]
syntax Stmt ::= Id "=" Exp ";"             [strict(2)]
| "if" "(" Exp ")" Stmt Stmt [strict(1)]
| "while" "(" Exp ")" Stmt
| "{" Stmt "}"                 [bracket]
| "{" "}"
> Stmt Stmt                      [left]
syntax Pgm  ::= "int" Ids ";" Stmt
syntax Ids  ::= List{Id, ","}
endmodule
module IMP
imports IMP-SYNTAX
imports DOMAINS
syntax KResult ::= Int
configuration
<T> <k> $PGM:Pgm </k> <state> .Map </state> </T>
rule <k> X:Id => I ...</k>
<state>... X |-> I ...</state>
rule I1 + I2 => I1 +Int I2
rule I1 - I2 => I1 -Int I2
rule <k> X = I:Int; => . ...</k>
<state>... X |-> (_ => I) ...</state>
rule S1:Stmt S2:Stmt => S1 ~> S2          [structural]
rule if (I) S _ => S requires I =/=Int 0
rule if (0) _ S => S
rule while(B) S
=> if(B) {S while(B) S} {}              [structural]
rule {} => .                              [structural]
rule <k> int (X, Xs => Xs); S </k>
<state> ... (. => X |-> 0) </state>  [structural]
rule int .Ids; S => S                     [structural]
endmodule
```

Fig. 6. A complete definition for the language IMP

The module `IMP` defines the operational semantics of IMP in terms of a set of human-readable rewrite rules (followed by the keyword `rule`). The category `KResult` tells $\mathbb{K}$ which categories contain non-reducible values. It helps $\mathbb{K}$ perform efficiently with evaluation contexts. The only category of values here is `Int`. Configuration is a core concept in the $\mathbb{K}$ framework. A *configuration* represents a *program execution state*, holding all information that is needed for program execution. Configurations are organized into *cells*, which are labeled and can be nested. Simple languages such as IMP have only a few cells, while complex real languages such as C may have a lot more. Configurations are written in XML format.

The configurations of IMP have two cells: a k cell and a state cell. For clarity, we gather both cells and put them in a top-level cell called the T cell, but it is not mandatory. The k cell holds the

rest computation (program fragments) that needs to execute and the state cell holds a map from program variables to their values in the memory. Initially, theell holds the empty map, denoted as `.Map`. In $\mathbb{K}$, we write "`.`" for "nothing", and `.Map` means that nothing has type `Map`.

Initially, the `k` cell contains an IMP program `$PGM:Pgm`, where `$PGM` is a special $\mathbb{K}$ variable name that tells $\mathbb{K}$ the program is saved in a source file, and the name of the file is passed as argument in the command line when $\mathbb{K}$ is invoked. $\mathbb{K}$ will then read the source file and parse it as a `Pgm`, and put the result in the `k` cell.

$\mathbb{K}$ defines the language semantics in terms of a set of rewrite rules. A rewrite rule has the form *lhs* => *rhs*, saying that any configuration γ that matches *lhs* rewrites to *rhs*, but as we will see later, $\mathbb{K}$ offers a more flexible and succinct way to define rewrite rules. All rewrite rules in a language definition specify a transition system on *configurations*, giving an operational semantics of the language. Notice that rewrites rules are inherently nondeterministic and concurrent, which makes it easy and naturally to define semantics for nondeterministic/concurrent languages in $\mathbb{K}$.

We emphasize two important characteristics of rewrites rules in $\mathbb{K}$. The first is *local rewrites*, i.e., the rewrite symbol "=>" does not need to appear in the top level, but can appear locally in which the rewrite happens. Take as an example the rule that looks up the value of a program variable in the state. Instead of writing

```
rule <k> X:Id ... </k> <state> ... X |-> I ... </state>
  => <k> I    ... </k> <state> ... X |-> I ... </state>
```

we can write the rewrite locally as

```
rule <k> X:Id => I ... </k> <state> ... X |-> I ... </state>
```

to not only reduce space but also avoid duplicates. The "`...`" has a special meaning in $\mathbb{K}$. It stands for things "that exist but do not change in the rewrite". The rule, therefore, says that if a program variable `X:Id` is in the top of the computation in the `k` cell, and `X` binds to the integer `I` somewhere in theell, then rewrite `X:Id` to its value `I`, and do not change anything else.

The second characteristic of rewrite rules in $\mathbb{K}$ is *configuration inference and completion*. The rewrite rules may not explicitly mention all cells in the configuration, but only mention related ones. $\mathbb{K}$ will infer the implicit cells and compete the configuration automatically. For example, instead of writing

```
rule <T> <k> I1 + I2 => I1 +Int I2 ... </k>
         <state> M </state> </T>
```

one can simply write `rule I1 + I2 => I1 +Int I2` which is simpler. It is also more modular: if in the future we need to add a new cell to the configuration, then we do not need to modify the rules above, as the new cells can be inferred and completed by $\mathbb{K}$ automatically. In fact, configuration inference and completion is one of the most important features that make $\mathbb{K}$ definitions extensible and easy to adapt to language changes.

The rest of the semantics are self-explained. The rule for assignment `X = I:Int;.` updates the value that is bound to `X` in theell, as specified in the local rewrite `X |-> (_ => I)`. Here the underscore "`_`" is an anonymous $\mathbb{K}$ variable. After the update, the assignment statement `X = I:Int;` is removed from the `k` cell, as specified by the local rewrite `X = I:Int; => .`. Recall that the dot "`.`" means nothing, and rewriting something to a dot means removing it. Attribute `structural` means the associated rewrite rule is not counted as an explicit step by $\mathbb{K}$, but an implicit (quite) one. It should not affect how $\mathbb{K}$ executes the programs. The empty statement `{}` simply reduces to nothing. The last two rules process the declaration list of program variables and initialize their values to zero.

5.2 Contexts

Matching logic allows us to define a very general notion of context. Our contexts can be used not only to define evaluation strategies of various language constructs, like how evaluation contexts are traditionally used [**?**], but also for configuration abstraction to enhance modularity and reuse, and for matching multiple sub-patterns of interest at the same time.

Like λ in Section **??**, contexts are also defined as binders. However, they are defined as schemas parametric in the sorts of their hole and result, respectively, and their application is controlled by their structure and surroundings. We first define the generic infrastructure for contexts:

$Context\{s, s'\}$	sort schema, where s (hole sort) and s' (result sort) range over any sorts
$\gamma^0\{s, s'\} \in \Sigma_{s \times s', Context\{s,s'\}}$	symbol schema, for all sorts s, s'
$\gamma\_.\_\{s, s'\}(\Box{:}s, T{:}s') \equiv \exists \Box.\gamma^0\{s, s'\}(\Box, T)$	here, $\Box$ is an ordinary variable
$\_[\_]\{s, s'\} \in \Sigma_{Context\{s,s'\} \times s, s'}$	symbol schema, for all sorts s, s'
$\_[\_]\{s, s\}(\gamma\_.\_\{s, s\}(\Box{:}s, \Box), T{:}s) = T$	axiom schema for identity contexts, for all sorts s

And the following axiom schema for composing nested contexts:

$$\begin{aligned}&\_[\_]\{s', s''\}(C_1{:}Context\{s', s''\}, \_[\_]\{s, s'\}(C_2{:}Context\{s, s'\}, T{:}s))\\ =&\_[\_]\{s, s''\}(\gamma\_.\_\{s, s''\}(\Box{:}s, \_[\_]\{s', s''\}(C_1{:}Context\{s', s''\}, \_[\_]\{s, s'\}(C_2{:}Context\{s, s'\}, \Box{:}s))), T{:}s)\end{aligned}$$

The sort parameters of axiom schemas can usually be inferred from the context. To ease notation, from here on we assume they can be inferred and apply the mixfix notation for symbols containing "_" in their names. With these, the last two axiom schemas above become:

$$(\gamma\Box.\ \Box)[T] = T$$
$$C_1[C_2[T]] = (\gamma\Box.C_1[C_2[\Box]])[T]$$

We write $C_1 \circ C_2 \equiv \gamma\Box.C_1[C_2[\Box]]$ for nested context. Using this notation, the last axiom becomes

$$C_1[C_2[T]] = (C_1 \circ C_2)[T]$$

The above sort, symbol and axiom schemas are generic and tacitly assumed in all definitions that make use of contexts. Let us now illustrate specific uses of contexts.

5.2.1 Evaluation Strategies of Language Constructs. Suppose that a programming language has an if-then-else statement, say $ite \in \Sigma_{BExp \times Stmt \times Stmt, Stmt}$, whose evaluation strategy is to first evaluate its first argument and then, depending on whether it evaluates to *true* or *false*, to rewrite to either its second argument or its third argument. We here only focus on its evaluation strategy and not its reduction rules; the latter will be discussed in Section **??**. Assuming that all reductions/rewrites apply in context, as discussed in Section **??**, we can state that *ite* is given permission to apply reductions within its first argument with the following axiom:

$$ite(T, S_1, S_2) = (\gamma\Box.\ ite(\Box, S_1, S_2))[T]$$

In practice, the above equation is often oriented as two rewrite rules. The rewrite rule from left to right is called *heating rule*, and the one from right to left is called *cooling rule*. In addition to sort/parameter inference, front-ends of implementations of matching logic are expected to provide shortcuts for such rather boring axioms. For example, $\mathbb{K}$ provides the strict attribute to be used with symbol declarations for exactly this purpose; for example, the evaluation strategy of *ite*, or the axiom above, is defined with the attribute strict(1) associated to the declaration of the symbol *ite*.

As an example, suppose that besides *ite* with strategy strict(1) we also have an infix operation $\_ < \_ \in \Sigma_{BExp \times BExp, BExp}$ with strategy strict(1,2) (i.e., it has two axioms like above, corresponding to

each of its two arguments). Using these axioms, we can infer the following:

$$\begin{array}{rcll} ite(1 < x, S_1, S_2) & = & (\gamma\Box.ite(\Box, S_1, S_2))[1 < x] & \text{// strict(1) axiom for } ite \\ & = & (\gamma\Box.ite(\Box, S_1, S_2))[(\gamma\Box.1 < \Box)[x]] & \text{// strict(2) axiom for } \_ < \_ \\ & = & \gamma\Box.((\gamma\Box.ite(\Box, S_1, S_2))[(\gamma\Box.1 < \Box)[\Box]])[x] & \text{// axiom for nested context} \end{array}$$

Notice that γ is a binder, so the last pattern above is alpha-equivalent to

$$\gamma\Box_1.((\gamma\Box_2.ite(\Box_2, S_1, S_2))[(\gamma\Box_3.1 < \Box_3)[\Box_1]])[x]$$

in which we rename all placeholder variables to prevent confusion. Again, we use strictness axioms, but this time from right to left, and simplify the above pattern as follows:

$$\begin{array}{rcll} & & \gamma\Box_1.((\gamma\Box_2.ite(\Box_2, S_1, S_2))[(\gamma\Box_3.1 < \Box_3)[\Box_1]])[x] & \\ & = & \gamma\Box_1.((\gamma\Box_2.ite(\Box_2, S_1, S_2))[1 < \Box_1])[x] & \text{// strict(2) axiom for } \_ < \_ \\ & = & \gamma\Box_1.(ite(1 < \Box_1, S_1, S_2))[x] & \text{// strict(1) axiom for } ite \end{array}$$

Therefore, $ite(1 < x, S_1, S_2)$ can be matched against a pattern of the form $C[x]$, where C, as expected, is $\gamma\Box_1.(ite(1 < \Box_1, S_1, S_2))$, a context of sort $Context\{BExp, Stmt\}$. That is, x has been "pulled" out of the *ite* context. We point out that the above is a typical example of reasoning about contexts in matching logic. One applies a sequence of strictness axioms, from left to right, to pull out the redex. Then, the resulting nested contexts can be combined and merged to one uniform context with the axiom for nested context. Finally, the body of the uniform context can be simplified by applying the sequence of strictness axiom from right to left.

Now other semantic rules or axioms can be applied to reduce x, by simply matching x in a context. At any moment during the reduction, the axioms above can be applied backwards and thus whatever x reduces to can be "plugged" back into its context. This way, the axiomatic approach to contexts in matching logic achieves the "pull and plug" mechanism underlying reduction semantics with evaluation contexts [**?**] by means of logical deduction using the generic sound and complete proof system in Section **??**. Also, notice that our notion of context is more general than that in reduction semantics. That is, it is not only used for reduction or in order to isolate a redex to be reduced, but it can be used for matching any relevant data from a program configuration. More examples below will illustrate that.

5.2.2 Multi-Hole Contexts and Configuration Abstraction. Contexts with multiple holes can also be easily supported by our approach, also without anything extra but the already existing deductive system of matching logic. A notation for multi-hole context application, however, is recommended in order to make patterns easier to read. The way we define multi-hole contexts in matching logic is similar to how we define multi-arity functions in lambda calculus by curring them. Specifically,

$$(\gamma\Box_1\Box_2\ldots\Box_n.\ T)[T_1, T_2, \ldots, T_n] \equiv (\gamma\Box_n.\ \ldots(\gamma\Box_2.\ (\gamma\Box_1.\ T)[T_1])[T_2]\ldots)[T_n]$$

Although $\gamma\Box_1\Box_2\ldots\Box_n.\ T$ correspond to no patterns, we take a freedom to call them *multi-hole contexts* and let meta-variables C range over them, i.e., we take the freedom to write $C[T_1, T_2, \ldots, T_n]$. Notice that we require placeholder variables $\Box_1, \ldots, \Box_n$ to be different, and $T_1, \ldots, T_n$ should not have free occurrences of any placeholder variables. We believe multi-hole contexts can be formalized as patterns, but we have not found any need for it yet.

Multi-hole contexts are particularly useful to define abstractions over program configurations. Indeed, $\mathbb{K}$ provides and promotes *configuration abstraction* as a mechanism to allow compact and modular language semantic definitions. The idea is to allow users to only write the parts of the program configuration that are necessary in semantic rules, the rest of the configuration being inferred automatically. This configuration abstraction process that is a crucial and distinctive feature of $\mathbb{K}$ can be now elegantly explained with multi-hole contexts.

To make the discussion concrete, suppose that we have a program configuration (cfg) that contains the code (k), the environment (env) mapping program variables to locations, and a memory (mem) mapping locations to values. For example, the term/pattern

$$\mathsf{cfg}(\mathsf{k}(\mathit{ite}(1 < x, S_1, S_2); S_3),\ \mathsf{env}(x \mapsto l, R_{env}),\ \mathsf{mem}(l \mapsto a, R_{mem}))$$

denotes a configuration containing the program "$\mathit{ite}(1 < x, S_1, S_2); S_3$" that starts with an *ite* statement followed by the rest of the program S_3, the environment "$x \mapsto l, R_{env}$" holding a binding of x to location l and the rest of the bindings R_{env}, and the memory "$l \mapsto a, R_{mem}$" holding a binding of l to value a and the rest of the bindings R_{mem}. In $\mathbb{K}$, in order to replace x in the program above with its value a by applying the lookup semantic rule, we need to match the configuration above against the pattern $C[x, x \mapsto l, l \mapsto a]$, where C is a multi-hole context. First, like we did with the strictness axiom of *ite*, we need to give contextual matching permission to operate in the various places of the configuration where we want to match patterns in context. In our case, we do that everywhere:

$$\begin{aligned}
&\mathsf{cfg}(T, E, M) = (\gamma\Box.\ \mathsf{cfg}(\Box, E, M))[T]\\
&\mathsf{cfg}(K, T, M) = (\gamma\Box.\ \mathsf{cfg}(K, \Box, M))[T]\\
&\mathsf{cfg}(K, E, T) = (\gamma\Box.\ \mathsf{cfg}(K, E, \Box))[T]\\
&\mathsf{k}(T) = (\gamma\Box.\ \mathsf{k}(\Box))[T]\\
&\mathsf{env}(T) = (\gamma\Box.\ \mathsf{env}(\Box))[T]\\
&\mathsf{mem}(T) = (\gamma\Box.\ \mathsf{mem}(\Box))[T]
\end{aligned}$$

Additionally, we also give permission for contextual matchings to take place in maps, which are regarded as patterns built with the infix pairing construct "$\_ \mapsto \_$" and an associative and commutative merge operation, here denoted as a comma "$\_, \_$", which has additional properties which are not relevant here:

$$(M_1, M_2) = (\gamma\Box.\ (M_1, \Box))[M_2] = (\gamma\Box.\ (\Box, M_2))[M_1]$$

The following matching logic proof shows how the configuration above can be transformed so that it can be matched by a multi-hole context as discussed:

$$\begin{aligned}
&\mathsf{cfg}(\mathsf{k}(\mathit{ite}(1 < x, S_1, S_2); S_3),\ \mathsf{env}(x \mapsto l, R_{env}),\ \mathsf{mem}(l \mapsto a, R_{mem})) &=\\
&(\gamma\Box_3.\ \mathsf{cfg}(\mathsf{k}(\mathit{ite}(1 < x, S_1, S_2); S_3),\ \mathsf{env}(x \mapsto l, R_{env}),\ \mathsf{mem}(\Box_3, R_{mem})))[l \mapsto a] &=\\
&(\gamma\Box_3.\ (\gamma\Box_2.\ \mathsf{cfg}(\mathsf{k}(\mathit{ite}(1 < x, S_1, S_2); S_3),\ \mathsf{env}(\Box_2, R_{env}),\ \mathsf{mem}(\Box_3, R_{mem})))[x \mapsto l])[l \mapsto a] &=\\
&(\gamma\Box_2\Box_3.\ \mathsf{cfg}(\mathsf{k}(\mathit{ite}(1 < x, S_1, S_2); S_3),\ \mathsf{env}(\Box_2, R_{env}),\ \mathsf{mem}(\Box_3, R_{mem})))[x \mapsto l, l \mapsto a] &=\\
&(\gamma\Box_2\Box_3.\ (\gamma\Box_1.\ \mathsf{cfg}(\mathsf{k}(\mathit{ite}(1 < \Box_1, S_1, S_2); S_3),\ \mathsf{env}(\Box_2, R_{env}),\ \mathsf{mem}(\Box_3, R_{mem})))[x])[x \mapsto l, l \mapsto a] &=\\
&(\gamma\Box_1\Box_2\Box_3.\ \mathsf{cfg}(\mathsf{k}(\mathit{ite}(1 < \Box_1, S_1, S_2); S_3),\ \mathsf{env}(\Box_2, R_{env}),\ \mathsf{mem}(\Box_3, R_{mem})))[x, x \mapsto l, l \mapsto a]
\end{aligned}$$

Here, the first, the second, and the fourth equations are by context reasoning as in Section 5.2.1 while the third and the fifth equations are by multi-hole context definition. Therefore, the configuration pattern

$$\mathsf{cfg}(\mathsf{k}(\mathit{ite}(1 < x, S_1, S_2); S_3),\ \mathsf{env}(x \mapsto l, R_{env}),\ \mathsf{mem}(l \mapsto a, R_{mem}))$$

can be matched by a pattern $C[x, x \mapsto l, l \mapsto a]$, where C is a multi-hole context. Sometimes configurations can be much more complex than the above, in which case one may want to make use of nested contexts to disambiguate. For example, the pattern

$$C_{\mathsf{cfg}}[\mathsf{k}(C_{\mathsf{k}}[x]), \mathsf{env}(C_{\mathsf{env}}[x \mapsto l]), \mathsf{mem}(C_{\mathsf{mem}}[l \mapsto a])]$$

makes it clear that x, $x \mapsto l$, and $l \mapsto a$ must be located inside the k, end, and mem configuration semantic components, or cells, respectively.

Remind what $\mathbb{K}$ is, and say that we now have a semantics of $\mathbb{K}$. Define contexts, show axioms of contexts in the monad style. And then give a few $\mathbb{K}$ rules as examples. Have 1 page about (symbolic) execution example, and 1 page about verification example.

Talk about parametric sorts. Say that we have a semantics of $\mathbb{K}$. Show axioms for contexts in the monad style. Show two examples. Show strictness axioms.

6 RELATED WORK (1 PAGE)

7 CONCLUSION AND FUTURE WORK (1 PAGE)