The Semantics of $\mathbb{K}$

Formal Systems Laboratory
University of Illinois at Urbana-Champaign

November 7, 2017

Please feel free to contribute to this report in all ways. You could add new contents, remove redundant ones, refactor and organize the texts, and correct typos, but please follow the FSL rules for editing, though; e.g., <80 characters per line, each sentence on a new line, etc.

1 Introduction

$\mathbb{K}$ is a best effort realization of matching logic [16]. Matching logic allows us to mathematically define arbitrarily infinite theories, which are not in general possible to describe finitely. $\mathbb{K}$ proposes a finitely describable subset of matching logic theories. Since its inception in 2003 as a notation within Maude [2] convenient for teaching programming languages [15], until recently $\mathbb{K}$'s semantics was explained either by translation to rewriting logic [12] or by translation to some form of graph rewriting [18]. These translations not only added clutter and came at a loss of part of the intended meaning of $\mathbb{K}$, but eventually turned out to be a serious limiting factor in the types of theories and languages that could be defined. Matching logic was specifically created and developed to serve as a logical, semantic foundation for $\mathbb{K}$, after almost 15 years of experience with using $\mathbb{K}$ to define the formal semantics of real-life programming languages, including C [5, 9], Java [1], JavaScript [13], Python [8, 14], PHP [7], EVM [11, 10].

Matching logic allows us to define *theories* (S, Σ, A) consisting of potentially infinite sets of *sorts* S, of *symbols* Σ over sorts in S (also called S-symbols), and of *patterns* A built with symbols in Σ (also called Σ-patterns), respectively, and provides models that interpret the symbols relationally, which in turn yield a *semantic validity* relation $(S, \Sigma, A) \models \varphi$ between theories (S, Σ, A) and Σ-patterns φ. Matching logic also has a Hilbert-style complete proof system, which allows us to derive new patterns φ from given theories (S, Σ, A), written $(S, \Sigma, A) \vdash \varphi$. When the sorts and signature are understood, we omit them; for example, the completeness of matching logic then states that for any matching logic theory (S, Σ, A) and any Σ-pattern φ, we have $A \models \varphi$ iff $A \vdash \varphi$.

...

Will add more here as we finalize the notation. we need some convincing example. Maybe parametric maps?

2 Matching Logic

In this section we first recall basic matching logic syntax and semantics notions from [16] at a theoretical level. Then we discuss schematic/parametric ways to finitely define infinite matching logic theories. Finally, we introduce theoretical foundations underlying the notion of "builtins".

We should make this paper self-contained in terms of definitions and only refer to [16] for details and proofs. We want people who trust [16] to not have to leave this paper in order to understand the semantics of $\mathbb{K}$. So we should define signatures, symbols, patterns, models, satisfaction, proof system, etc.

2.1 Syntax

Assume a matching logic *signature* (S, Σ), where S is the set of its *sorts* and Σ is the set of its *symbols*. When S is understood, we write just Σ for a signature instead of (S, Σ). Assume a set *Name* of infinitely many *variable names*. We partition Σ in sets $\Sigma_{s_1 \ldots s_n, s}$ of symbols of *sorting signature* $s_1 \ldots s_n, s$, where $s_1, \ldots, s_n, s \in S$. The formulae of matching logic are called *patterns*, although we may also call them *formulae*. Patterns of sort $s \in S$ are generated by the following grammar:

$$
\begin{array}{rl}
\varphi_s ::= & x : s \quad \text{where } x \in \mathit{Name} \text{ and } s \in S \\
\mid & \sigma(\varphi_{s_1}, ..., \varphi_{s_n}) \quad \text{where } \sigma \in \Sigma_{s_1 \ldots s_n, s} \text{ and } \varphi_{s_1}, ..., \varphi_{s_n} \text{ of appropriate sorts} \\
\mid & \varphi_s \wedge \varphi_s \\
\mid & \neg \varphi_s \\
\mid & \exists x : s'. \varphi_s \quad \text{where } x \in N \text{ and } s' \in S
\end{array}
$$

Figure 1: The grammar of matching logic patterns. For each $s \in S$, φ_s are *patterns of sort* s.

Instead of writing $x : s$, we write just x for a variable when s is understood from the contexts. In Figure 1, the "wedge $\wedge$", "negation $\neg$", and "exists $\exists$" are in fact parametric over the sort s:

$$\text{"wedge } \wedge_s\text{"}, \text{ "negation } \neg_s\text{"}, \text{ and "exists } \exists_s\text{"},$$

but we often omit the subscripted s if it is understood from the contexts. Given a symbol $\sigma \in \Sigma_{s_1 \ldots s_n, s}$, $s_1 \ldots s_n$ are called the *argument sorts* of σ, and s is called the *return sort* of σ. n is called the arity of σ. When $n = 0$, we call σ a *constant symbol* or simply a *constant*. If σ is a constant, we write the pattern just σ instead of $\sigma()$ for simplicity.

Let PATTERN be the S-sorted set of patterns. The grammar above can be infinite, i.e., can have infinitely many non-terminals and productions, because S and Σ can be infinite. Also, as usual, it only defines the syntax of formulae and not their semantics. For example, patterns $x \wedge y$ and $y \wedge x$ are distinct elements in the language of the grammar, in spite of them being semantically/provably equal in matching logic. For notational convenience, we take the liberty to use mix-fix syntax for operators in Σ and parentheses for grouping. Also, we take the freedom to suffix variables with their sort preceded by a colon whenever we want to clarify their sort in a pattern. For example, if $\mathit{Nat} \in S$ and $\_ + \_, \_ * \_ \in \Sigma_{\mathit{Nat} \times \mathit{Nat}, \mathit{Nat}}$ then we may write "$(x : \mathit{Nat} + y : \mathit{Nat}) * z : \mathit{Nat}$" instead of "$\_ * \_(\_ + \_(x, y), z)$ and $x, y, z \in \mathit{Nat}$". More notational conventions will be introduced along the way as we use them. We adopt the following derived constructs ("syntactic sugar"):

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

We adapt from first-order logic the notions of *free variable* ($FV(\varphi)$ is the set of free variables of φ) and of variable-capture-free *substitution* ($\varphi[\varphi'/x]$ denotes φ whose free occurrences of x are replaced with φ', possibly renaming bound variables in φ to avoid capturing free variables of φ').

A matching logic *theory* is a triple (S, Σ, A) where (S, Σ) is a signature and A is a set of patterns called *axioms*. When S is understood or not important, we write (Σ, A) instead of (S, Σ, A).

2.2 Semantics and Basic Properties

A *matching logic* (S, Σ)*-model* M consists of: An S-sorted set $\{M_s\}_{s \in S}$, where each set M_s, called the *carrier of sort* s *of* M, is assumed non-empty; and a function $\sigma_M : M_{s_1} \times \cdots \times M_{s_n} \to \mathcal{P}(M_s)$ for each symbol $\sigma \in \Sigma_{s_1 \ldots s_n, s}$, called the *interpretation* of σ in M. It is important to note that in matching logic symbols are interpreted as functions into power-set domains, that is, as *relations*, and not as usual functions like in FOL. We tacitly use the same notation σ_M for its extension to argument sets, $\mathcal{P}(M_{s_1}) \times \cdots \times \mathcal{P}(M_{s_n}) \to \mathcal{P}(M_s)$. When S is understood we may call M a Σ*-model*, and when both S and Σ are understood we call it simply a *model*. We let $Mod(S, \Sigma)$, or $Mod(\Sigma)$ when S is understood, denote the (category of) models of a signature (S, Σ). Given a model M and a map $\rho : Var \to M$, called an M*-valuation*, let its extension $\overline{\rho} : \textsc{Pattern} \to \mathcal{P}(M)$ be inductively defined as follows:

- $\overline{\rho}(x) = \{\rho(x)\}$, for all $x \in Var_s$
- $\overline{\rho}(\sigma(\varphi_1, \ldots, \varphi_n)) = \sigma_M(\overline{\rho}(\varphi_1), \ldots \overline{\rho}(\varphi_n))$ for all $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ and appropriate $\varphi_1, ..., \varphi_n$
- $\overline{\rho}(\neg\varphi) = M_s \setminus \overline{\rho}(\varphi)$ for all $\varphi \in \textsc{Pattern}_s$
- $\overline{\rho}(\varphi_1 \wedge \varphi_2) = \overline{\rho}(\varphi_1) \cap \overline{\rho}(\varphi_2)$ for all φ_1, φ_2 patterns of the same sort
- $\overline{\rho}(\exists x.\varphi) = \bigcup\{\overline{\rho'}(\varphi) \mid \rho' : Var \to M,\ \rho'{\restriction}_{Var \setminus \{x\}} = \rho{\restriction}_{Var \setminus \{x\}}\} = \bigcup_{a \in M} \overline{\rho[a/x]}(\varphi)$

where "$\setminus$" is set difference, "$\rho{\restriction}_V$" is ρ restricted to $V \subseteq Var$, and "$\rho[a/x]$" is map ρ' with $\rho'(x) = a$ and $\rho'(y) = \rho(y)$ if $y \neq x$. If $a \in \overline{\rho}(\varphi)$ then we say a *matches* φ (with witness ρ).

Pattern φ_s is an M*-predicate*, or a *predicate in* M, iff for any M-valuation $\rho : Var \to M$, it is the case that $\overline{\rho}(\varphi_s)$ is either M_s (it holds) or $\emptyset$ (it does not hold). Pattern φ_s is a *predicate* iff it is a predicate in all models M. For example, $\top_s$ and $\bot_s$ are predicates, and if φ, φ_1 and φ_2 are predicates then so are $\neg\varphi$, $\varphi_1 \wedge \varphi_2$, and $\exists x\,.\,\varphi$. That is, the logical connectives of matching logic preserve the predicate nature of patterns. Model M *satisfies* φ_s, written $M \models \varphi_s$, iff $\overline{\rho}(\varphi_s) = M_s$ for all $\rho : Var \to M$. Pattern φ is *valid*, written $\models \varphi$, iff $M \models \varphi$ for all M. If $A \subseteq \textsc{Pattern}$ then $M \models A$ iff $M \models \varphi$ for all $\varphi \in A$. A *entails* φ, written $A \models \varphi$, iff for each M, $M \models A$ implies $M \models \varphi$. We may subscript $\models$ with the signature whenever we feel it clarifies the presentation; that is, we may write $\models_{(S,\Sigma)}$ or $\models_\Sigma$ instead of $\models$. A *matching logic specification* is a triple (S, Σ, A), or just (Σ, A) when S is understood, with A a set of Σ-patterns. Given matching logic specification (Σ, A) we let $Mod(\Sigma, A)$, also denoted by $[\![(\Sigma, A)]\!]$ be its (category of) models $\{M \mid M \in Mod_\Sigma,\ M \models_\Sigma A\}$.

A signature (S', Σ') is called a *subsignature* of (S, Σ), written $(S', \Sigma') \hookrightarrow (S, \Sigma)$, if and only if $S' \subseteq S$ and $\Sigma' \subseteq \Sigma$. If $M \in Mod(\Sigma)$ then we let $M{\restriction}_{\Sigma'} \in Mod(\Sigma')$ denote its Σ'*-reduct*, or simply its *reduct* when Σ' is understood, defined as follows: $(M{\restriction}_{\Sigma'})_{s'} = M_{s'}$ for any $s' \in S'$ and $\sigma'_{M{\restriction}_{\Sigma'}} = \sigma'_M$ for any $\sigma' \in \Sigma'$. It may help to think of signatures as interfaces and of models as *implementations* of such interfaces. Indeed, models provide concrete values for each sort, and concrete relations for symbols. Then the reduct $M{\restriction}_{\Sigma'}$ can be regarded as a "wrapper" of the implementation M of Σ turning it into an implementation of Σ', or a reuse of a richer implementation in a smaller context.

2.3 Useful Symbols and Notations

Matching logic is rather poor by default. For example, it has no functions, no predicates, no equality, and although symbols are interpreted as sets and variables are singletons, it has no membership or inclusion. All these operations are very useful, if not indispensable in practice. Fortunately, they and many others can be defined axiomatically in matching logic. That is, whenever we need these in order to define (the semantics of other symbols in) a matching logic specification (Σ, A), we can

add corresponding symbols to Σ and corresponding patterns to A as axioms, so that, in models, the desired symbols or patterns associated to the desired operations get interpreted as expected.

For any sorts $s_1, s_2 \in S$, assume the following *definedness* symbol and corresponding pattern:

$$\begin{array}{ll} \lceil \_ \rceil^{s_2}_{s_1} \in \Sigma_{s_1,s_2} & \textit{// Definedness symbol} \\ \lceil x{:}s_1 \rceil^{s_2}_{s_1} \in A & \textit{// Definedness pattern} \end{array}$$

Like in many logics, free variables are assumed universally quantified. So the definedness pattern axiom above should be read as "$\forall x{:}s_1 \,.\, \lceil x \rceil^{s_2}_{s_1}$". If S is infinite, then we have infinitely many definedness symbols and patterns above. It is easy to show that if $\varphi \in \textsc{Pattern}_{s_1}$ then $\lceil \varphi \rceil^{s_2}_{s_1}$ is a predicate which holds iff φ is defined: if $\rho : \mathit{Var} \to M$ then $\overline{\rho}(\lceil \varphi \rceil^{s_2}_{s_1})$ is either $\emptyset$ (i.e., $\overline{\rho}(\bot_{s_2})$) when $\overline{\rho}(\varphi) = \emptyset$ (i.e., φ undefined in ρ), or is M_{s_2} (i.e., $\overline{\rho}(\top_{s_2})$) when $\overline{\rho}(\varphi) \neq \emptyset$ (i.e., φ defined).

We also define *totality*, $\lfloor \_ \rfloor^{s_2}_{s_1}$, as a derived construct dual to definedness:

$$\lfloor \varphi \rfloor^{s_2}_{s_1} \quad \equiv \quad \neg \lceil \neg \varphi \rceil^{s_2}_{s_1}$$

Totality also behaves as a predicate. It states that the enclosed pattern is matched by all values. That is, if $\varphi \in \textsc{Pattern}_{s_1}$ then $\lfloor \varphi \rfloor^{s_2}_{s_1}$ is a predicate where if $\rho : \mathit{Var} \to M$ is any valuation then $\overline{\rho}(\lfloor \varphi \rfloor^{s_2}_{s_1})$ is either $\emptyset$ (i.e., $\overline{\rho}(\bot_{s_2})$) when $\overline{\rho}(\varphi) \neq M_{s_1}$ (i.e., φ is not total in ρ), or is M_{s_2} (i.e., $\overline{\rho}(\top_{s_2})$) when $\overline{\rho}(\varphi) = M_{s_1}$ (i.e., φ is total).

Equality can be defined quite compactly using pattern totality and equivalence. For each pair of sorts s_1 (for the compared patterns) and s_2 (for the context in which the equality is used), we define $\_ =^{s_2}_{s_1} \_$ as the following derived construct:

$$\varphi =^{s_2}_{s_1} \varphi' \qquad \equiv \qquad \lfloor \varphi \leftrightarrow \varphi' \rfloor^{s_2}_{s_1} \qquad \text{where } \varphi, \varphi' \in \textsc{Pattern}_{s_1}$$

Equality is also a predicate. if $\varphi, \varphi' \in \textsc{Pattern}_{s_1}$ and $\rho : \mathit{Var} \to M$ then $\overline{\rho}(\varphi =^{s_2}_{s_1} \varphi') = \emptyset$ iff $\overline{\rho}(\varphi) \neq \overline{\rho}(\varphi')$, and $\overline{\rho}(\varphi =^{s_2}_{s_1} \varphi') = M_{s_2}$ iff $\overline{\rho}(\varphi) = \overline{\rho}(\varphi')$.

Similarly, we can define *membership*: if $x \in \mathit{Var}_{s_1}$, $\varphi \in \textsc{Pattern}_{s_1}$ and $s_2 \in S$, then let

$$x \in^{s_2}_{s_1} \varphi \qquad \equiv \qquad \lceil x \wedge \varphi \rceil^{s_2}_{s_1}$$

Membership is also a predicate. Specifically, for any $\rho : \mathit{Var} \to M$, $\overline{\rho}(x \in^{s_2}_{s_1} \varphi) = \emptyset$ iff $\rho(x) \notin \overline{\rho}(\varphi)$, and $\overline{\rho}(x \in^{s_2}_{s_1} \varphi) = M_{s_2}$ iff $\rho(x) \in \overline{\rho}(\varphi)$.

Since s_1 and s_2 can usually be inferred from context, we write $\lceil \_ \rceil$ or $\lfloor \_ \rfloor$ instead of $\lceil \_ \rceil^{s_2}_{s_1}$ or $\lfloor \_ \rfloor^{s_2}_{s_1}$, respectively, and similarly for the equality and membership. Also, if the sort decorations cannot be inferred from context, then we assume the stated property/axiom/rule holds for all such sorts. For example, the generic pattern axiom "$\lceil x \rceil$ where $x \in \mathit{Var}$" replaces all the axioms $\lceil x{:}s_1 \rceil^{s_2}_{s_1}$ above for all the definedness symbols for all the sorts s_1 and s_2.

Refer to this later when we talk about parameters.

Note that, by default, symbols are interpreted as relations in matching logic. It is often the case, though, that we want symbols to be interpreted as *functions*. This can be easily done by axiomatically constraining those symbols to evaluate to singletons. For example, if f is a unary symbol, then the pattern equation "$\forall x \,.\, \exists y \,.\, f(x) = y$" (the convention for free variables allows us to drop the universal quantifier) states that in any model M, the set $f_M(a)$ contains precisely one element for any $a \in M$. Inspired from similar notations in other logics, we adopt the familiar notation "$\sigma : s_1 \times \cdots \times s_n \to s$" to indicate that σ is a symbol in $\Sigma_{s_1 \ldots s_n, s}$ and that the pattern $\exists y \,.\, \sigma(x_1, \ldots, x_n) = y$ is in A. In this case, we call σ a *function symbol* or even just a *function*. Patterns built with only function symbols are called *term patterns*, or simply just *terms*.

Should we discuss constructors, too?

2.4 Sound and Complete Deduction

Currently, our sound and complete proof system for matching logic extends that of first-order logic with equality with axioms and proof rules for membership. In order for this to be feasible, we need equality and membership, which are available when the definedness symbol is available, as seen in Section 2.3. Therefore, in this section we assume a matching logic theory (S, Σ, A) which includes all the definedness symbols discussed in Section 2.3 (and thus also totality, equality and membership). We conjecture that it is possible to craft a proof system that does not rely on the existence of definedness symbols. Nevertheless, that will not significantly change the $\mathbb{K}$ semantics approach taken in this paper, because all the existing axioms and proof rules will be proven as lemmas in the new proof system. Therefore, any proofs done with the proof system below will be easily translatable to proofs done with the new proof system, whenever the latter will be available.

Our proof system is a Hilbert-style proof system (not to be confused with a Gentzen-style proof system). To avoid any confusion about our notation and to remind the reader the basics of axiom and proof rule *schemas*, we start by briefly recalling what a Hilbert-style proof system is, but for specificity we do it in the context of matching logic. A *proof rule* is a pair $(\{\varphi_1, ..., \varphi_n\}, \varphi)$, written

$$\frac{\varphi_1 \quad \cdots \quad \varphi_n}{\varphi}$$

The formulae φ_1, ..., φ_n are called the *hypotheses* and φ is called the *conclusion* of the rule. The order in which the hypotheses occur in a proof rule is irrelevant. When $n = 0$ we call the proof rule an *axiom* and take the freedom to drop the empty hypotheses and the separating line, writing it simply as "φ". A proof system allows us to *formally prove* or *derive* formulae. Specifically, a proof system yields a *provability relation*, written $A \vdash \varphi$, for any given specification (Σ, A), defined inductively as follows: $A \vdash \varphi$ if $\varphi \in A$; and $A \vdash \varphi$ if there is a proof rule like above such that $A \vdash \varphi_1$, ..., $A \vdash \varphi_n$. Formulae in A can therefore be regarded as axioms, and we even take the freedom to call them axioms when there is no misunderstanding. However, note that a proof system is fixed for the target logic, including all its axioms (i.e., proof rules with no hypotheses). We use the notation $T \vdash_{\text{fin}} \varphi$ or $A \vdash_{\text{fin}} \varphi$ to emphasize the fact that the theory T or the set of axioms A is finite.

Proof systems can be and typically are infinite, that is, contain infinitely many proof rules. To write them using finite resources (space, time), we make use of *proof schemas* and *meta-variables*. As an example, let us recall the usual proof system of propositional logic, which is also included in the proof system we propose here for matching logic:

Propositional calculus proof rules:

1. $\varphi_1 \to (\varphi_2 \to \varphi_1)$ (PROPOSITIONAL$_1$)
2. $(\varphi_1 \to (\varphi_2 \to \varphi_3)) \to ((\varphi_1 \to \varphi_2) \to (\varphi_1 \to \varphi_3))$ (PROPOSITIONAL$_2$)
3. $(\neg\varphi_1 \to \neg\varphi_2) \to (\varphi_2 \to \varphi_1)$ (PROPOSITIONAL$_3$)
4. $\dfrac{\varphi_1 \quad \varphi_1 \to \varphi_2}{\varphi_2}$ (MODUS PONENS)

In propositional logic, φ_1, φ_2 and φ_3 above are meta-variables ranging over propositions. The first three are *axiom schemas* while the fourth is a proper *rule schema*. Schemas can be regarded

as templates, which specify infinitely many instances, one for each instance of the meta-variables. We take the four proof rule schemas of propositional logic unchanged and regard them as proof rule schemas for matching logic. Note, however, that the *meta-variables now range over patterns* of the same sort, and thus there is a schema for each sort.

Matching logic also includes the proof system of first-order logic with equality. However, as explained in [16], we prefer to replace the FOL substitution proof rule with two rules, called *functional substitution* and *functional variable* below:

First-order logic with equality proof rules:

5. $\vdash (\forall x \,.\, \varphi_1 \to \varphi_2) \to (\varphi_1 \to \forall x \,.\, \varphi_2)$ when $x \notin FV(\varphi_1)$ ($\forall$)

6. $\dfrac{\varphi}{\forall x \,.\, \varphi}$ (Universal Generalization)

7. $\vdash (\forall x \,.\, \varphi) \wedge (\exists y \,.\, \varphi' = y) \to \varphi[\varphi'/x]$ (Functional Substitution)

Need to define functional patterns.

8. $\exists y \,.\, x = y$ (Functional Variable)

9. $\varphi = \varphi$ (Equality Introduction)

10. $\varphi_1 = \varphi_2 \wedge \varphi[\varphi_1/x] \to \varphi[\varphi_2/x]$ (Equality Elimination)

In addition to the above rules borrowed from FOL with equality, matching logic also introduces the following rules for (reasoning about) membership.

Membership rules:

11. $\dfrac{\varphi}{\forall x \,.\, x \in \varphi}$ (Membership Introduction)

12. $\dfrac{\forall x \,.\, x \in \varphi}{\varphi}$ (Membership Elimination)

13. $x \in y = (x = y)$ when $x, y \in \mathit{Var}$ (Membership Variable)

14. $x \in \neg\varphi = \neg(x \in \varphi)$ (Membership $\neg$)

15. $x \in \varphi_1 \wedge \varphi_2 = (x \in \varphi_1) \wedge (x \in \varphi_2)$ (Membership $\wedge$)

16. $(x \in \exists y \,.\, \varphi) = \exists y \,.\, (x \in \varphi)$ when $x, y \in \mathit{Var}$ distinct (Membership $\exists$)

17. $x \in \sigma(\varphi_1, ..., \varphi_{i-1}, \varphi_i, \varphi_{i+1}, ..., \varphi_n) =$
$\exists y \,.\, (y \in \varphi_i \wedge x \in \sigma(\varphi_1, ..., \varphi_{i-1}, y, \varphi_{i+1}, ..., \varphi_n))$ (Membership Symbol)

The following result establishes the soundness and completeness of the proof system above:

Theorem 1. *[16] For any matching logic specification (Σ, A) and Σ-pattern φ, $A \models \varphi$ iff $A \vdash \varphi$.*

Note that Theorem 1 also holds when the matching logic specification is infinite, that is, when it has infinitely many sorts and symbols in Σ and infinitely many axioms in A.

3 Finite Mechanisms to Define Infinite Theories

The theoretical results discussed so far imposed no finiteness restrictions on the sets of sorts, symbols, or patterns that form a matching logic theory. In practice, however, like in many other logics or formalisms, we have to limit ourselves to finitely describable theories. The simplest approach to achieve that would be to require the theories to be finite; however, like in many other logics and formalisms, such a requirement would simply be too strong to be practical. Instead, we have to adopt or develop conventions, mechanisms and/or languages that allow us to describe potentially infinite theories using a finite amount of resources (paper, space, characters, etc.). For example, many logics allow *axiom schemas* as a way to finitely define infinite theories.

To prepare the ground for our proposal in Section **??**, we here discuss, informally, some ways to finitely describe infinite theories. Let us start with sorts. Suppose that we have a finite set of *basic sorts*, such as *Nat*, *Int*, *Bool*, etc. Below are several *sort schema* examples that allow us to extend the set of sorts with infinitely many new sorts:

$List\{s\}$	for any sort s
$Set\{s\}$	for any sort s
$Bag\{s\}$	for any sort s
$Bag_p\{s\}$	for any sort s which is not of the form $Bag_p\{\_\}$
$Map\{s, s'\}$	for any sorts s, s' // for (partial) maps from keys of sort s to values of sort s'
$Map_p\{s, s'\}$	for any sorts s, s' such that s is not of the form $Map_p\{\_, \_\}$
$Context\{s, s'\}$	for any sorts s, s' // for contexts with holes of sort s and results of sort s'

Sort schemas, like all schemas, have a *least fixed-point* interpretation; that is, they can be regarded as sort constructors which *inductively define* a (potentially) infinite set of sorts. The five sort schemas above together with the basic sorts, for example, generate infinitely many sorts, such as $List\{Nat\}$, $Bag\{List\{Nat\}\}$, $Map\{List\{Int\}, Set\{Int\}\}$, $Context\{Set\{Int\}, Map\{Int, Bool\}\}$, etc. Note that sort schemas, like all schemas, can have *side conditions*. For example, the schema $Bag_p\{s\}$ (of proper bags) disallows instances where s is already a proper bag. In general, to formally write side conditions over schema parameters we need access to a *meta-level*. We will do this in Section **??**, but for now we will continue to use side conditions informally.

Let us now move to symbols. Here are a few *symbol schemas*, defining infinitely many symbols:

Make sure we always use the same symbol for empty sequences of sorts.

$nil\{s\} \in \Sigma_{*, List\{s\}}$	for any sort s
$cons\{s\} \in \Sigma_{s \times List\{s\}, List\{s\}}$	for any sort s
$append\{s\} \in \Sigma_{List\{s\} \times List\{s\}, List\{s\}}$	for any sort s
$empty\{s, s'\} \in \Sigma_{*, Map\{s,s'\}}$	for any sorts s, s'
$bind\{s, s'\} \in \Sigma_{s \times s', Map\{s,s'\}}$	for any sorts s, s'
$merge\{s, s'\} \in \Sigma_{Map\{s,s'\} \times Map\{s,s'\}, Map\{s,s'\}}$	for any sorts s, s'

And here are some *pattern schemas*, each defining infinitely many patters:

$$\begin{array}{rl}
append\{s\}(nil\{s\}, l' : List\{s\}) =^{s'}_{List\{s\}} l' & \text{for any sorts } s, s' \\
append\{s\}(cons\{s\}(x : s, l : List\{s\}), l' : List\{s\}) \\
\quad cons\{s\}(x : s, append\{s\}(l, l')) & =^{s'}_{List\{s\}} \quad \text{for any sorts } s, s' \\
merge\{s, s'\}(empty\{s, s'\}, m : Map\{s, s'\}) =^{s''}_{Map\{s,s'\}} m & \text{for any sorts } s, s', s'' \\
merge\{s, s'\}(m : Map\{s, s'\}, m' : Map\{s, s'\}) \\
\quad merge\{s, s'\}(m' : Map\{s, s'\}, m : Map\{s, s'\}) & =^{s''}_{Map\{s,s'\}} \quad \text{for any sorts } s, s', s'' \\
\multicolumn{2}{c}{\ldots}
\end{array}$$

Note that all the sort, symbol and pattern schemas above are parametric in sorts only. In theory, they can be parameterized with anything, including with integer numbers, with symbols, and even with arbitrary patterns. We found it sufficient in practice to parameterize sort and symbol schemas with sort parameters only, so for the time being we do not consider any other parameters for these. However, pattern schemas sometimes need to be parameterized with symbols and patterns in addition to sorts. Consider, for example, the following pattern schema describing the important property of substitution when applied to a pattern rooted in a symbol:

$$\begin{array}{rl}
\sigma(\varphi_1, ..., \varphi_n)[\varphi/x : s'] =^{s''}_{s} \sigma(\varphi_1[\varphi/x], ..., \varphi_n[\varphi/x]) & \text{for any sorts } s, s', s'', s_1, ..., s_n, \\
& \text{any symbol } \sigma \in \Sigma_{s_1 \times ... \times s_n, s}, \\
& \text{and any pattern } \varphi \text{ of sort } s
\end{array}$$

The above pattern schema is parametric in sorts, symbols and patterns, and, of course, has infinitely many instances.

We have seen some simple side conditions in the examples of sort schemas above. Pattern schemas, however, tend to have more complex side conditions. Below are several other common examples of pattern schemas, some of them with nontrivial side conditions:

$$\begin{array}{rl}
\varphi[\varphi_1/x : s] \wedge (\varphi_1 =^{s'}_{s} \varphi_2) \rightarrow \varphi[\varphi_2/x : s] & \text{where } s, s' \text{ are any sorts, } \varphi \text{ any pattern of} \\
& \text{sort } s', \text{ and } \varphi_1, \varphi_2 \text{ any patterns of sort } s \\
\forall x : s.\varphi \rightarrow \varphi[t/x] & \text{where } s \text{ is any sort and } t \text{ is any } \textit{syntactic pattern}, \text{ or } \textit{term}, \\
& \text{or sort } s, \text{ i.e., one containing only variables and symbols} \\
\varphi_1 \texttt{ + } \varphi_2 = \varphi_1 +_{Nat} \varphi_2 & \text{where } \varphi, \varphi' \text{ are } \textit{ground} \text{ syntactic patterns of sort } \textit{Nat}, \\
& \text{that is, patterns built only with symbols } \texttt{zero} \text{ and } \texttt{succ} \\
(\varphi_1 \rightarrow \varphi_2) \rightarrow (\varphi[\varphi_1/x] \rightarrow \varphi[\varphi_2/x]) & \text{where } \varphi \text{ is a } \textit{positive context in } x, \text{ that is, a pattern} \\
& \text{containing only one occurrence of } x \text{ with no negation } (\neg) \\
& \text{on the path to } x, \text{ and where } \varphi_1, \varphi_2 \text{ are any patterns} \\
& \text{having the same sort as } x
\end{array}$$

One of the major goals of this paper is to propose a formal language and an implementation of it that allow us to finitely specify potentially infinite matching logic theories presented with finitely many sort, symbol and pattern schemas.

4 Important Case Studies

In this section we illustrate the power of matching logic by showing how it can handle binders, fixed-points, contexts, and rewriting and reachability. These important notions or concepts can be defined as syntactic sugar or as particular theories in matching logic, so that the uniform matching logic proof system in Section 2.4 can be used to reason about all of these. In particular, $\mathbb{K}$ can now be given semantics fully in matching logic. That is, a $\mathbb{K}$ language definition becomes a matching logic theory, and the various tools that are part of the $\mathbb{K}$ framework become best-effort implementations of targeted proof search using the deduction system in Section 2.4.

4.1 Binders

The $\mathbb{K}$ framework allows to define binders, such as the λ binder in λ-calculus, using the attribute `binder`. But what does that really mean? Matching logic appears to provide no support for binders, except for having its own binder, the existential quantifier $\exists$. Here we only discuss untyped λ-calculus, but the same ideas apply to any calculus with binders.

Suppose that S consists of only one sort, *Exp*, for λ-expressions. Although matching logic provides an infinite set of variables Var_{Exp} of sort *Exp*, we cannot simply define $\lambda\_.\_$ as a symbol in $\Sigma_{Var_{Exp} \times Exp, Exp}$, for at least two reasons: first, Var_{Exp} is *not* an actual sort at the core level (as seen in Section 7, it is a sort at the meta-level); second, we want the first argument of $\lambda\_.\_$ to bind its occurrences in the second argument, and symbols do not do that. To ease notation, from here on in this section assume that all variables are in Var_{Exp} and all patterns have sort *Exp*.

The key observation here is that the $\lambda\_.\_$ construct in λ-calculus *performs two important operations*: on the one hand it builds a binding of its first argument into its second, and one the other hand it builds a term. Fortunately, matching logic allows us to separate these two operations, and use the builtin existential quantifier for binding. Specifically, we define a symbol λ^0 and regard λ as syntactic sugar for the pattern that existentially binds its first argument into λ^0:

$$\lambda^0 \in \Sigma_{Exp \times Exp, Exp}$$
$$\lambda x.e \equiv \exists x.\lambda^0(x, e)$$

Therefore, $\lambda^0(x, e)$ builds a term (actually a pattern) with no binding relationship between the its first argument x and other occurrences of x in term/pattern e, and then the existential quantifier $\exists x$ adds the binding relationship. Mathematically, we can regard λ^0 as constructing points (input, output) on the graph of the function, and then the existential quantifier gives us their union as stated by its matching logic semantics, that is, the actual function. Note that this same construction does not work in FOL, because there quantifiers apply to predicates and not to terms/patters. It is the very nature of matching logic to not distinguish between function and predicate symbols that makes the above work. The application can be defined as an ordinary symbol:

Explain why λ is not a symbol, but an alias

$$\_\ \_ \in \Sigma_{Exp \times Exp, Exp}$$

Let us now discuss the axiom patterns. First, note that we get the α-equality property,

$$\lambda x.e = \lambda y.e[y/x]$$

essentially for free, because matching logic's builtin existential quantifier and substitution already enjoy the α-equivalence property [16]. The β-equality, on the other hand, requires an important

side condition. To start the discussion, let us suppose that we naively define it as follows:

$$(\lambda x.e)e' = e[e'/x] \qquad \text{for any } \textit{pattern } e \qquad \text{// this is actually wrong!}$$

The problem is that e and e' cannot be just any arbitrary patterns. For example, if we pick e to be $\top$ and e' to be $\bot$, then we can show that $(\lambda x.\top)\bot = \bot$ (see Section 2.2: the interpretation of $\_\ \_$ is empty when any of its arguments is empty), and since $\top[\bot/x] = \top$ we get $\top = \bot$, that is, inconsistency. Matching logic, therefore, provides patterns that were not intended for λ-calculus. The solution is to restrict, with side conditions, the application of β-equality to only patterns that correspond to λ-terms in the original calculus:

$$(\lambda x.e)e' = e[e'/x] \qquad \begin{array}{l}\text{where } e,\ e' \text{ are patterns constructed only with variables}\\ \lambda \text{ binders (via desugaring) and application symbols}\end{array}$$

That is, we first identify a syntactic fragment of the universe of patterns which is in a bijective correspondence with the original syntactic terms of λ-calculus, and then restrict the application of the β-equality rule to only patterns in that fragment.

The above gives us an embedding of λ-calculus as a theory in matching logic. We conjecture that this embedding is a *conservative extension*, that is, if e and e' are two λ-terms, then $e = e'$ holds in the original λ-calculus if and only if the corresponding equality between patterns holds in the matching logic theory. The "only if" part is easy, because equational reasoning is sound for matching logic [16], but the "if" part appears to be non-trivial.

4.2 Fixed Points

Similarly to the λ-binder in Section 4.1, we can add a fixed-point μ-binder as follows:

$$\begin{array}{rl}\mu^0 \in \Sigma_{Exp\times Exp, Exp} & \\ \mu x.e \equiv \exists x.\mu^0(x, e) & \\ \mu x.e = e[\mu x.e/x] & \begin{array}{l}\text{where } e \text{ is a pattern corresponding to a term}\\ \text{in the original calculus (i.e., constructed with variables,}\\ \lambda \text{ and } \mu \text{ binders (via desugaring), and application symbols)}\end{array}\end{array}$$

Given any model M and interpretation $\rho : Var \to M$, pattern e yields a function $e_M : M \to \mathcal{P}(M)$ where $e_M(a) = \overline{\rho[a/x]}(e)$. It is easy to see that its point-wise extension $e_M : \mathcal{P}(M) \to \mathcal{P}(M)$ is monotone w.r.t. $\subseteq$, so by the Knaster-Tarski theorem it has a fixed point[1]. In fact, if we let X be the set $\overline{\rho}(\mu x.e)$, then the equation of μ above yields $X = e_M(X)$, that is, X is a fixed point of $e_M : \mathcal{P}(M) \to \mathcal{P}(M)$. We want, however, X to be the *least* fixed point of e_M. We do not know if that is possible, because M and ρ are arbitrary and many of the fixed-points of e_M may very well be unreachable with patterns. However, from a practical perspective, are those fixed points of any importance at all? Considering that when we do proofs we can only derive patterns, it makes sense to limit ourselves to only fixed points that can be expressed syntactically. Within this limited universe, we can axiomatize the *least fixed-point* nature of $\mu x.e$ with the following pattern schema:

$$e[e'/x] = e' \to \lfloor \mu x.e \to e' \rfloor \qquad \begin{array}{l}\text{where } e \text{ and } e' \text{ are patterns corresponding to terms}\\ \text{in the original calculus}\end{array}$$

[1] Moreover, the set of fixed-points of $e_M : \mathcal{P}(M) \to \mathcal{P}(M)$ forms a complete lattice.

It is now a simple exercise to add a dual, greatest fixed-point construct:

$$\begin{array}{rl}
\nu^0 \in \Sigma_{Exp \times Exp, Exp} & \\
\nu x.e \equiv \exists x.\nu^0(x, e) & \\
\nu x.e = e[\nu x.e/x] & \text{where } e \text{ is a pattern corresponding to a term in the original calculus} \\
e[e'/x] = e' \to \lfloor e' \to \nu x.e \rfloor & \text{where } e \text{ and } e' \text{ are patterns corresponding to terms in the original calculus}
\end{array}$$

An alternative is to define $\nu x.e$ directly as the dual of $\mu x.e$, that is, $\nu x.e \equiv \neg\mu x.\neg e$. Can we prove the pattern above them?

We extend our conjecture in Section 4.1 and conjecture that the embedding above, in spite of not necessarily yielding the expected absolute least fixed-points in all models (but least only relatively to patterns that can be constructed with the original syntax), remains a conservative extension: if e and e' are two terms built with the syntax of the original calculus, then $e = e'$ holds in the original calculus if and only if the corresponding equality holds in the corresponding matching logic theory. Like before, the "only if" part is easy.

4.3 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 [6], but also for configuration abstraction to enhance modularity and reuse, and for matching multiple sub-patterns of interest at the same time.

Brandon: how about the locality principle?

Like λ in Section 4.1, 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:

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

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 axiom schema above becomes:

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

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.

4.3.1 Evaluation Strategies of Language Constructs

Suppose that a programming language has an if-then-else statement, say $\mathsf{ite} \in \Sigma_{BExp \times Stmt \times Stmt, Stmt}$, 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 4.4. Assuming that all reductions/rewrites apply in context, as discussed in Section 4.4, we can state that ite is given permission to apply reductions within its first argument with the following axiom:

$$\mathsf{ite}(C[T], S_1, S_2) = (\gamma\ \Box\ .\ \mathsf{ite}(C[\Box], S_1, S_2))[T]$$

In particular, C can be the identity context, "$\gamma\ \Box\ .\ \Box$". 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 $\_ < \_ \in \Sigma_{AExp \times AExp, BExp}$ 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:

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

Therefore, $\mathsf{ite}(1 < x, S_1, S_2)$ can be matched against a pattern of the form $C[x]$, where C is a context of sort $\mathit{Context}\{\mathit{AExp}, \mathit{Stmt}\}$. That is, x has been "pulled" out of the ite context; 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 [6] by means of logical deduction using the generic sound and complete proof system in Section 2.4. 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.

4.3.2 Nested Contexts

Nesting of contexts comes for free in our approach, that is, nothing but reasoning using the deduction system of matching logic needs to be done in order to achieve matching with nested contexts. For example:

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

Therefore, $\mathsf{ite}(1 < x, S_1, S_2)$ can also be matched against a pattern of the form $C_1[C_2[x]]$, where C_1 is a context of sort $\mathit{Context}\{\mathit{BExp}, \mathit{Stmt}\}$ and C_2 of sort $\mathit{Context}\{\mathit{AExp}, \mathit{BExp}\}$. More interestingly,

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

Therefore, $\mathsf{ite}(1 < x, S_1, S_2)$ can also be matched against a pattern of the form $(\gamma\ \square\ .\ C_1[T_2])[x]$, where C_1 is a context of sort $\mathit{Context}\{\mathit{BExp}, \mathit{Stmt}\}$ and $C_2 = \gamma\ \square\ .\ T_2$ is a context of sort $\mathit{Context}\{\mathit{AExp}, \mathit{BExp}\}$. Notice that we cannot naively apply a context to another context, e.g., $C_1[C2]$, because the sorts do not match. Once the hole is explicitly mentioned as a binder in a context, what we really mean by $C_1[C2]$ is in fact $\gamma\ \square\ .\ C_1[T_2]$, where $C_2 = \gamma\ \square\ .\ T_2$.

4.3.3 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. Specifically,

$$(\gamma\ \square_1\ \square_2 \ldots \square_n\ .\ T)[T_1, T_2, \ldots, T_n] \equiv (\gamma\ \square_n\ .\ \ldots (\gamma\ \square_2\ .\ (\gamma\ \square_1\ .\ T)[T_1])[T_2] \ldots)[T_n]$$

Although $\gamma\square_1\square_2 \ldots \square_n.\ T$ 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 $C[T_1, T_2, \ldots, T_n]$. 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 abstracton 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

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

denotes a configuration containing the program "$\mathsf{ite}(1 < x, S_1, S_2);\ S_3$" 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:

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

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:

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

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:

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

Therefore, the configuration pattern

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

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.

4.4 Rewriting and Reachability

Matching logic patterns generalize terms: indeed, each term t is a particular pattern built only with variables and symbols. Furthermore, reachability rules in reachability logic [17, 3], which have the form $\varphi \Rightarrow \varphi'$ where φ and φ' are patterns of the same sort, generalize rewrite rules. A set of reachability rules $\mathcal{S}$, for example a $\mathbb{K}$ language definition, yields a binary relation $\_ \rightarrow_{\mathcal{S}} \_ \subseteq M \times M$ on any given model M, called a *transition system*. The transition system $(M, \rightarrow_{\mathcal{S}})$ is a mathematical model of $\mathcal{S}$, comprising all the dynamic behaviors of $\mathcal{S}$. Reachability logic consists of a proof system for reachability rules, which is sound and relatively complete. Until now, $\mathbb{K}$'s semantics was best explained in terms of reachability logic. However, it turns out that matching logic is expressive enough to represent both rewriting and reachability. Together with the other results above, this implies that matching logic can serve as a standalone semantic foundation of $\mathbb{K}$.

Let us first note that, in matching logic, giving a binary relation $R_s \in M_s \times M_s$ on the carrier of sort s of a model M is equivalent to interpreting a unary symbol $\circ \in \Sigma_{s,s}$ in M: indeed, for any $a, b \in M_s$, take $a\ R_s\ b$ iff $a \in \circ_M(b)$. If the intuition for $a\ R_s\ b$ is "state a transitions to state b", then the intuition for $\circ_M$ is that of "transition to" or "next": $\circ_M(b)$ is the set of states that

transition to b, or which next go to b. Next consider two patterns φ and φ' of sort s, and the pattern $\varphi \to \circ\varphi'$. We have $M \models \varphi \to \circ\varphi'$ iff $\overline{\rho}(\varphi) \subseteq \circ_M(\overline{\rho}(\varphi'))$ for any M-valuation ρ, iff for any $a \in \overline{\rho}(\varphi)$ there is some $b \in \overline{\rho}(\varphi')$ such that $a \; R_s \; b$, which is precisely the interpretation of a rewrite rule $\varphi \Rightarrow \varphi'$ in M. Hence, we can add multi-sorted rewriting to a matching logic theory as follows:

$$\begin{array}{rl} \circ\{s\} \in \Sigma_{s,s} & \text{// a "next" symbol schema} \\ \varphi \Rightarrow\{s\} \varphi' \equiv \varphi \to \circ\{s\}\varphi' & \text{// a "rewrite relation" alias schema} \end{array}$$

To avoid clutter, we write $\Rightarrow$ instead of $\Rightarrow\{s\}$ and assume that s can be inferred from context. If necessary, we may add a superscript n to indicate the number of rewrite steps; for example, $\varphi \Rightarrow^1 \varphi'$ means "φ rewrites to φ' in one step". By default, from here on $\Rightarrow$ means $\Rightarrow^1$.

We can now write semantic rules, like in $\mathbb{K}$. For example, for the configuration in Section 4.3.3, the rule for variable lookup can be as follows:

$$C[(x \Rightarrow a), x \mapsto l, l \mapsto a]$$

Or, if more structure in the context is needed or preferred:

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

Although irrelevant here, it is worth noting that the $\mathbb{K}$ frontend provides syntactic sugar for avoiding writing the contexts. Specifically, users can use ellipses "..." to "fill in the obvious context". For example, $\mathbb{K}$ frontends may allow users to write the two rules above as

$$x \Rightarrow a \ldots x \mapsto l \ldots l \mapsto a$$

and, respectively,

$$\mathsf{k}(x \Rightarrow a \ldots) \; \mathsf{env}(\ldots x \mapsto l \ldots) \; \mathsf{mem}(\ldots l \mapsto a \ldots)$$

Note that the top-level context is skipped, because there is always a top-level context and thus there is no need to mention it in each rule. A more interesting example is the rule for assignment in an imperative language, taking an assignment $x := b$ to skip and at the same time updating x in the memory. Using the compact notation above, this rule becomes:

$$(x := b) \Rightarrow \mathsf{skip} \ldots x \mapsto l \ldots l \mapsto (a \Rightarrow b)$$

There are two rewrites taking place at the same time in the rule above: one in the k cell rewriting $x := b$ to skip, and another one in the location of x in the memory rewriting whatever value was there, a, to the assigned value, b.

There are two important aspects left to explain before we can use $\mathbb{K}$ definitions to reason about programs. First, we need to be able to lift rewrites from inside patterns to the top level. Indeed, the rules above do not explain how an actual configuration that matches the lookup or the assignment pattern above transits to the next configuration. For that reason, we add axiom schemas to the matching logic theory that lift the $\circ$ symbol:

$$\sigma(*\varphi_1, \ldots, *\varphi_n) \to \circ\, \sigma(\varphi_1, \ldots, \varphi_n) \qquad \begin{array}{l} \text{where } \sigma \in \Sigma_{s_1 \times \cdots s_n, s}, \\ \varphi_1 \text{ is a pattern of sort } s_1, \\ \ldots, \\ \varphi_n \text{ is a pattern of sort } s_n, \\ \text{and at least one of } * \text{ is } \circ \end{array}$$

Together with structural framing [16], these axioms allow not only to lift multiple local rewrites that are part of the same rule into one rewrite higher in the pattern, but also to combine multiple parallel rewrites into one rewrite, thus giving natural support for *true concurrency*. Note that the axioms above allow to lift the $\circ$ symbol from any proper subset of orthogonal subpaterns, without enforcing the lifting of *all* the $\circ$ symbols. In other words, an *interleaving model of concurrency* is also supported. One can choose one or another methodologically, the underlying logic not enforcing any. We believe that this is the ideal scenario wrt concurrency.

The other important aspect that needs to be explained in order to allow full reasoning based on $\mathbb{K}$ definitions, is reachability. Specifically, we need a way to define reachability claims/specifications in matching logic. Thanks to matching logic's support for fixed-points, we can, in fact, define various LTL-, CTL-, or CTL*-like constructs. We leave most of these as exercise to the interested reader, here only showing how to define two of them which are useful to define and reason about reachability:

$$\begin{array}{rcll} \Diamond_{Strong}\varphi & \equiv & \mu f.(\varphi \vee \circ f) & \text{// strong eventually on one path} \\ \Diamond_{Weak}\varphi & \equiv & \nu f.(\varphi \vee \circ f) & \text{// weak eventually on one path} \end{array}$$

The pattern $\Diamond_{Strong}\varphi$ is matched by those states/configurations which eventually reach a state/configuration that matches φ, using the transition system associated to the unary symbol $\circ$ as described above. The pattern $\Diamond_{weak}\varphi$, one the other hand, is matched by those states/configurations which either never terminate or otherwise eventually reach a state/configuration that matches φ. With these, it is not hard to see that the (partial correctness) one-path reachability relation $\varphi \Rightarrow^{\exists} \varphi'$ of reachability logic [4] can be semantically captured by the pattern $\varphi \rightarrow \Diamond_{weak}\varphi'$. We leave it as a (non-trivial) exercise to the reader to also define the all-path reachability relation $\varphi \Rightarrow^{\forall} \varphi'$ and to prove all the proof rules of reachability logic as lemmas/theorems using the more basic proof system of matching logic.

5 Built-ins

It is rarely the case in practice that a matching logic theory, for example a programming language semantics, is defined from scratch. Typically, it makes use of built-ins, such as natural/integer/real numbers. While sometimes builtins can be defined themselves as matching logic theories, for example Booleans, in general such definitions may require sets of axioms which are not r.e., and thus may be hard or impossible to encode regardless of the chosen formalism. Additionally, different tools may need to regard or use the builtins differently; for example, an interpreter may prefer the builtins to be hooked to a fast implementation as a library, a symbolic execution engine may prefer the builtins to be hooked to an SMT solver like Z3, while a mechanical program verifier may prefer a rigorous definition of builtins using Coq, Isabelle, or Agda.

Recall from Section 2.2 that the semantics of a matching logic theory (Σ, A) was loosely defined as the collection of all Σ-models satisfying its axioms: $[\![(\Sigma, A)]\!] = \{M \mid M \in Mod_\Sigma,\ M \models_\Sigma A\}$. To allow all the builtin scenarios above and stay as abstract as possible w.r.t. builtins, we generalize matching logic theories and their semantics as follows.

Definition 2. A *matching logic theory with builtins* $(S_{builtin}, \Sigma_{builtin}, S, \Sigma, A)$, written as a triple $(\Sigma_{builtin}, \Sigma, A)$ and called just a *matching logic theory* whenever there is no confusion, is an ordinary matching logic theory together with a *subsignature of builtins* $(S_{builtin}, \Sigma_{builtin}) \hookrightarrow (S, \Sigma)$. Sorts in $S_{builtin} \subseteq S$ are called *builtin sorts* and symbols in $\Sigma_{builtin} \subseteq \Sigma$ are called *builtin symbols*.

Therefore, signatures identify a subset of sorts and symbols as builtin, with the intuition that implementations are now *parametric* in an implementation of their builtins. Or put differently, the semantics of a matching logic theory with builtins is parametric in a model for its builtins:

Definition 3. Given a matching logic theory with builtins $(\Sigma_{builtin}, \Sigma, A)$ and a *model of builtins* $B \in Mod(\Sigma_{builtin})$, we define the *$B$-semantics* of $(\Sigma_{builtin}, \Sigma, A)$ loosely as follows:

$$[\![(\Sigma_{builtin}, \Sigma, A)]\!]_B = \{M \mid M \in Mod_\Sigma,\ M \models_\Sigma A,\ M{\restriction_{\Sigma_{builtin}}} = B\}$$

We may drop B from B-semantics whenever the builtins model is understood from context.

Note that A may contain axioms over $\Sigma_{builtin}$, which play a dual role: they filter out the candidates for the models of builtins on the one hand, and they can be used in reasoning on the other hand. Theoretically, we can always enrich A with the set of *all* patterns matched by B, and thus all the properties of the model of builtins are available for reasoning in any context, but note that in practice there may be no finite or algorithmic way to represent those (e.g., the set of properties of the "builtin" model of natural numbers is not r.e.).

For this, we may want to organize ML as an institution.

6 Meta-Theory and Reflection in Matching Logic

As a follow-up to Section 3 where we introduced various *schema mechanisms* that allow us to define infinite theories using a finite amount of resources, we are going to give such schema mechanisms *a formal semantics*. We will define a matching logic theory called *the meta-theory* K as a reflective theory of matching logic in Section 7.

Reflection in logic and programming languages has been extensively studied in literature. We here quote a rather broad definition by Brian Smith, who in [19] defines reflection as

> An entitys integral ability to represent, operate on, and otherwise deal with its self in the same way that it represents, operates on and deals with its primary subject matter.

Following this definition, the meta-theory K is said to be a reflective one because (1) K itself is a matching logic theory and (2) one can use K to study and discuss the properties of matching logic and matching logic theories, as shown in the next theorem (the faithfulness theorem):

$$\frac{T \vdash \varphi}{K \cup [\![T]\!] \vdash_{\text{fin}} \texttt{\#provable}([\![\varphi]\!])}\ \text{(Upward Reflection)} \qquad \frac{K \cup [\![T]\!] \vdash_{\text{fin}} \texttt{\#provable}([\![\varphi]\!])}{T \vdash \varphi}\ \text{(Downward Reflection)}$$

where

- $[\![T]\!]$, called the *meta-representation of* T, consists of a finite number of new symbols and axioms depending on T;
- $K \cup [\![T]\!]$, called the *meta-theory instantiated by* T, is the meta-theory K plus the finitely many symbols and axioms in $[\![T]\!]$;
- $[\![\varphi]\!]$ is the *meta-representation of* φ as a matching logic pattern in $K \cup [\![T]\!]$;
- `#provable` is a symbol in K which axiomatizes the probability relation in matching logic (see Section 7.7)

The double bracket $[\![\_]\!]$ is called *lifting operation*, which brings objects to their meta-representations in the meta-theory K. Lifting operation will be defined in detail in Section 8, after we present in full length the meta-theory K in Secion 7

7 Meta-Theory of Matching Logic

In this section, we present a finite matching logic theory $K = (S_K, \Sigma_K, A_K)$ as *the meta-theory of matching logic*. The meta-theory K has a finite set S_K of sorts, a finite set Σ_S of symbols, and a finite set A_K of axioms. We will define the meta-theory K as a *a syntactic theory*, as we will explicitly enumerate all the finitely many axioms in the meta-theory K. In fact, we will provide a summary of the meta-theory K at the end of this section (see Section 7.8). We prefer a syntactic definition for the meta-theory K because we are mainly interested in doing reasoning in matching logic, and a purely syntactic definition helps us to develop a prover for matching logic. We will find the meta-theory K a model when we prove the faithfulness theorem in Section 10, in which we will introduce *the canonical model of the meta-theory*. However, this is not the emphasis of this section, and let us focus on the syntactic definition of K for now.

7.1 Warming-Up

The meta-theory K is a complex theory, and we will use many notations and conventions to help us define it. Some notations and naming conventions are specific to the meta-theory K and they will be defined when they are firstly used. The others are general ones and are not specific to K. In this subsection, we provide an overview of all tricks that we play in defining K.

7.1.1 Variable Names

Recall that the matching logic grammar that we introduced in Figure 1 assumes a set *Name* of infinitely many variable names. Generally speaking, it does not matter what sets of variable names we choose, but good names are often more intuitive and helpful than bad ones. In a complex theory like K, picking good names can be extremely important.

In K, we assume the following letters and their variants (primed, subscripted, etc.) are in the set *Name* of variable names:

$$Name = \{x, y, z, s, f, S, L, u, v, \sigma, \varphi, \psi, \dots\}$$

Starting from Section 7.2, the only mathematical variable or meta-variable that we will use is $\#s$. We use $\#s$ to denote any sort in S_K. Readers may notice that φ and ψ are now normal variable names like x and y, and *not* meta-variables for patterns as in Section 3. This is intended.

7.1.2 Three Ways of Defining Predicate Symbols

Recall that predicate symbols in matching logic are symbols whose interpretations in models are mappings which return either the empty set $\emptyset$ or the total set on all inputs. We can also define predicate symbols in a syntactic way. A symbol $\sigma \in \Sigma_{s_1 \dots s_n, s}$ is said to be a predicate symbol in a theory Γ if

$$\Gamma \vdash \forall x_1 \dots x_n.((\sigma(x_1, \dots, x_n) = \top) \vee (\sigma(x_1, \dots, x_n) = \bot))$$

In terms of defining predicate symbols, there are in general three approaches. We use the predicate symbol *isEmpty* as an example to illustrate all three approaches. The predicate symbol *isEmpty* checks whether a list is empty or not. The key thing is that we want to use *isEmpty* in any sort contexts.

One way to do that is to introduce a special sort *Pred* and define

$$isEmpty \in \Sigma_{List,Pred}$$

and two axioms

$$isEmpty(nil) \qquad\qquad \neg isEmpty(cons(x, L))$$

Now *isEmpty* is defined as a predicate symbol but it has a specific return sort *Pred*, and therefore it cannot be used in any sort context. In order to use *isEmpty* in any sort context, we have to introduce the following abbreviation:

$$b \equiv (b = \top_{Pred}) \qquad \text{for any pattern } b \text{ of sort } Pred$$

and the equality lets us use it in any sort context.

The second approach is to define

$$isEmpty\colon List \to Bool$$

as a function from *List* to *Bool*, with two axioms:

$$isEmpty(nil) = true \qquad\qquad isEmpty(cons(x, L)) = false$$

Now *isEmpty* is defined as a function and has a specific return sort *Bool*. To use it in any sort context, we have to introduce the following abbreviation:

$$b \equiv (b = true) \qquad \text{for any pattern } b \text{ of sort } Bool$$

and the equality lets us use it in any sort context.

The last approach, which is also the approach that we adopt in defining K, is to define a set of predicate symbols

$$isEmpty\{s\} \in \Sigma_{List,s} \qquad \text{for any sort } s$$

with two axiom schemas:

$$isEmpty\{s\}(nil) \qquad\qquad \neg isEmpty\{s\}(cons(x, L))$$

Now *isEmpty* is defined as a *symbol schema*, and it can be put in any sort context by instantiating the schema with the corresponding sort.

It is mostly a taste of flavor in choosing which approach to use. The third approach might seem a bit verbose but it is the most fundamental and essential one, as it does not require equality and thus it can be used in all theories even if the definedness symbols are not defined.

7.1.3 Using Symbol Schemas Does Not Make Meta-Theory Infinite

As we have shown in Section 7.1.2, the meta-theory K contains some predicate symbols defined using symbol schemas. Smart readers might be wondering whether that leads to an infinite theory, and the answer is no. Using symbol schemas in defining the meta-theory K does not make it an infinite theory, because we never use *sort schemas*. Therefore, the meta-theory K only has a finite number of sorts. As long as the number of sorts are finite, we can freely use both symbol schemas and axiom schemas to help us defining K without taking the risk of making the whole theory an infinite one, because any (symbol or axiom) schema will have only finitely many instances.

7.1.4 Typewriter Fonts versus Italic Fonts

As we have seen, variable names in K are printed in italic fonts. The only meta-variable *#s* that we use is also printed in italic fonts. All sorts and symbols in K are printed in typewriter fonts, and most of them start with a sharp "`#`".

7.1.5 Uppercase versus Lowercase

As a general principle, all sorts in K start with an uppercase letter while all symbols in K start with a lowercase letter.

7.1.6 The Meta-Theory Is Extended By Need

The meta-theory is our answer to the question "what is the semantics of $\mathbb{K}$?". It represents our approach to support reflection in matching logic. On the other hand, we never regard the meta-theory as a final product. As $\mathbb{K}$ evolves, the meta-theory also has to evolve accordingly.

7.1.7 A Roadmap Of The Definition Of Meta-Theory

Starting from the next subsection, we will define the meta-theory K in full detail. We will organize the lengthy definition into six sections (running from Section 7.2 to Section 7.7) with each subsection defining a high-cohesion-loose-coupling part of the meta-theory:

Section	Contents
Section 7.2	Definition of characters and strings
Section 7.3	Definition of meta-representations of sorts and symbols
Section 7.4	Definition of finite lists
Section 7.5	Definition of meta-representations of patterns
Section 7.6	Definition of predicate symbols that define theories
Section 7.7	Axiomatization of the proof system of matching logic
Section 7.8	A summary of all sorts, symbols, and axioms in K

We tried to make sure for any $x, y \in \{2, 3, 4, 5, 6, 7\}$ and $x < y$, that Section 7.x does not refer to the contents that are introduced in Section 7.y, but it is not always the case. The readers should be prepared to see some sorts and symbols are used before they are formally defined, in which case there will be informal explanations about those sorts and symbols.

7.2 Characters and Strings

The sort `#Char` is the sort for *characters*. It has the following $26 + 26 + 10 + 3 = 65$ functional constructors:

```
'a': → #Char   'b': → #Char   'c': → #Char
     ...            ...            ...
'x': → #Char   'y': → #Char   'z': → #Char
'A': → #Char   'B': → #Char   'C': → #Char
     ...            ...            ...
'X': → #Char   'Y': → #Char   'Z': → #Char
'0': → #Char        ...       '9': → #Char
'#': → #Char   '\': → #Char   ''': → #Char
```

Strings as Finite Lists of Characters. Characters are used to construct *strings*, defined as cons-lists of characters. The sort for strings is `#CharList`, which will be defined in Section 7.4. The sort `#CharList` has two functional constructors

$$\texttt{\#nilCharList}: \to \texttt{\#CharList}$$
$$\texttt{\#consCharList}: \texttt{\#Char} \times \texttt{\#CharList} \to \texttt{\#CharList}$$

Notation 4. It is a bit inconvenient and heavy to treat strings as cons-lists of characters, and it is not quite user-friendly to write `#CharList` for the sort of strings. Therefore, we introduce `#String` as an alias of `#CharList`, and we write `#epsilon` as an alias of `#nilCharList` that represents the empty string. As a convention, strings are often represented by texts wrapped with quotation marks. For example, instead of writing

$$\texttt{\#consCharList}(\text{`}\texttt{a}\text{'}, \texttt{\#consCharList}(\text{`}\texttt{b}\text{'}, \texttt{\#consCharList}(\text{`}\texttt{c}\text{'}, \texttt{\#epsilon})))$$

we simply write "`abc`" with the double-quotation marks.

7.3 Matching Logic Sorts and Symbols

The sort `#Sort` is the sort for matching logic sorts. It has one functional constructor

$$\texttt{\#sort}: \underbrace{\texttt{\#String}}_{\text{name}} \times \underbrace{\texttt{\#SortList}}_{\text{sort parameters}} \to \texttt{\#Sort}$$

The constructor `#sort` takes a string as the *sort name* and a list of sorts as the *sort parameters*, and constructs the corresponding sort. The sort `#SortList` is the sort for cons-lists of sorts and it is defined in Section 7.4. For parametric sorts such as $List\{Nat\}$ and $Map\{Nat,Bool\}$, it is easy to tell what their sort names are and what their sort parameters are, and therefore it is easy to see how to construct their meta-representations in K using the constructor `#sort`. For non-parametric sort such as Nat, we regard it as the "parametric sort" $Nat\{\}$ which takes zero sort parameter. This unified view of parametric and non-parametric sorts helps to keep the meta-theory as simple as possible. We will adopt the same view in dealing with parametric symbols and non-parametric symbols, too.

The sort `#Symbol` is the sort for matching logic symbols. It has one functional constructor

$$\texttt{\#symbol}: \underbrace{\texttt{\#String}}_{\text{name}} \times \underbrace{\texttt{\#SortList}}_{\text{sort parameters}} \times \underbrace{\texttt{\#SortList}}_{\text{argument sorts}} \times \underbrace{\texttt{\#Sort}}_{\text{return sort}} \to \texttt{\#Symbol}$$

Since the argument sorts and the return sort of a symbol are included in its meta-representation, it is often called the meta-representation of "decorated symbols", where the word "decorated" means that not only the name and sort parameters of a symbol are included, but also its argument sorts and return sort. For non-parametric symbols such as *zero* and *plus*, we regard them as the "parametric symbols" $zero\{\}$ and $plus\{\}$ which take zero sort parameter.

Two useful getter functions for `#Symbol` are defined:

$$\texttt{\#getArgumentSorts}: \texttt{\#Symbol} \to \texttt{\#SortList}$$
$$\texttt{\#getReturnSort}: \texttt{\#Symbol} \to \texttt{\#Sort}$$

Both getter functions have very simple axioms

$$\forall f \forall S \forall S' \forall s.\texttt{\#getArgumentSorts}(\texttt{\#symbol}(f, S, S', s)) = S'$$
$$\forall f \forall S \forall S' \forall s.\texttt{\#getReturnSort}(\texttt{\#symbol}(f, S, S', s)) = s$$

Recall that we often write just x instead of $x\texttt{:}s$ for logic variables when the sort s is understood from the context. Here we are writing

f	as a shorthand of $f\texttt{:\#String}$	which is a variable of sort `#String`
S	as a shorthand of $S\texttt{:\#SortList}$	which is a variable of sort `#SortList`
S'	as a shorthand of $S'\texttt{:\#SortList}$	which is a variable of sort `#SortList`
s	as a shorthand of $s\texttt{:\#Sort}$	which is a variable of sort `#Sort`

The universal quantifiers "$\forall f \forall S \forall S' \forall s$" in the front of both axioms are not necessary and can be removed without any change in semantics meaning. In later sections, we will often not write such unnecessary universal quantifiers unless there is a chance to confuse logic variables with mathematical variables or meta-variables. In that case, we will explicitly write those universal quantifiers to emphasize certain variables are not meta-variables but normal logic variables, because only logic variables can be quantified. Readers will see such examples in Section 7.5.

Meta-Representations of Definedness Symbols Definedness symbols, also known as the *ceiling* symbol, are required by the sound and complete proof system of matching logic [16]. For any matching logic theory and two sorts s, s' in the theory, we assume there exists a symbol $\lceil \_ \rceil_s^{s'}$ with the axiom $\lceil x\texttt{:}s \rceil_s^{s'}$ defined in the theory. The symbol $\lceil \_ \rceil_s^{s'}$ will have its unique meta-representation in the meta-theory K, constructed using the constructor `#symbol`.

Notice that $\lceil \_ \rceil_s^{s'}$ is a 2-dimensional representation of the symbol $ceil\{s, s'\}$:

$$\lceil \_ \rceil_s^{s'} \equiv ceil\{s, s'\}$$

In the following, we will interchangeably use both the 2-dimensional representation and the linear representation.

We introduce a function (not a constructor) `#‘ceil` in the meta-theory K:

$$\texttt{\#‘ceil}\texttt{:} \quad \underbrace{\texttt{\#Sort}}_{\text{the subscripted sort } s} \quad \times \quad \underbrace{\texttt{\#Sort}}_{\text{the superscripted sort } s'} \quad \rightarrow \texttt{\#Symbol}$$

which helps to construct the meta-representation of the definedness symbol $ceil\{s, s'\}$. The function `#‘ceil` is *not* a constructor of sort `#Symbol` but rather a helper function, as shown in the next axiom:

$$\forall s \forall s'.(\texttt{\#‘ceil}(s, s') = \texttt{\#symbol}(\text{“}\texttt{ceil}\text{”}, (s, s'), (s), s'))$$

Here we are writing

s	as a shorthand of $s\texttt{:\#Sort}$
s'	as a shorthand of $s'\texttt{:\#Sort}$
(s, s')	as a shorthand of $\texttt{\#consSortList}(s, \texttt{\#consSortList}(s', \texttt{\#nilSortList}))$
(s)	as a shorthand of $\texttt{\#consSortList}(s, \texttt{\#nilSortList})$

The last two shorthands are defined in Notation 6.

Remark 5. The function `#‘ceil` is used to construct the meta-representations of definedness symbols *in the object theories*. Smart readers may already notice that our meta-theory K has equality, and thus must have definedness symbols defined. That is exactly true. In fact, the meta-theory K has n^2 different definedness symbols:

$$\lceil \_ \rceil^{\#s'}_{\#s} \in \Sigma_K \qquad \text{for any } \#s, \#s' \in S_K \text{ are sorts in the meta-theory } K$$

where $n = |S_K|$ is the number of sorts in the meta-theory. The function `#‘ceil` $\in \Sigma_K$ has nothing to do with those n^2 definedness symbols in the meta-theory K. It is simply a direct correspondence of the definedness symbol $ceil\{s, s'\}$ in the object theories.

Here is one more word about where the weird name `#‘ceil` comes from. The first letter sharp `#` tells us that it belongs to the meta-theory K. The second letter quot ‘ tells us that it corresponds to something in object theories (in this case, the definedness symbols in object theories). As a general principle, we use the quot ‘ for naming meta-theory symbols which directly correspond to things in object theories, so that we will not have a naming conflict with other meta-theory symbols.

7.4 Finite Lists

As we have seen, the meta-theory K has a sort `#SortList` for cons-lists of sorts, and a sort `#CharList` for cons-lists of characters. In fact, the meta-theory K defines the following five different types of cons-lists:

Sort	Meaning
`#CharList`	Sort for cons-lists of characters `#Char`
`#SortList`	Sort for cons-lists of sorts `#Sort`
`#SymbolList`	Sort for cons-lists of symbols `#Symbol`
`#VariableList`	Sort for cons-lists of variables `#Variable`
`#PatternList`	Sort for cons-lists of patterns `#Pattern`

We have defined the sorts `#Char`, `#Sort`, and `#Symbol`. The sorts `#Variable` and `#Pattern` will be defined in Section 7.5.

All five types of cons-lists share a similar definition. In the following, we only show to define cons-lists of sorts as an example. The definition for the rest four types of cons-lists should be constructed in the same way.

7.4.1 Finite Lists of Sorts

The sort `#SortList` is the sort for cons-lists of sorts. It has two functional constructors:

$$\texttt{\#nilSortList}: \to \texttt{\#SortList}$$
$$\texttt{\#consSortList}: \texttt{\#Sort} \times \texttt{\#SortList} \to \texttt{\#SortList}$$

The function

$$\texttt{\#appendSortList}: \texttt{\#SortList} \times \texttt{\#SortList} \to \texttt{\#SortList}$$

takes two lists and returns the concatenation of them. Assume s is a `#Sort` variable, and S_0, S are `#SortList` variables, the following two axioms are defined

$$\texttt{\#appendSortList}(\texttt{\#nilSortList}, S) = S$$
$$\texttt{\#appendSortList}(\texttt{\#consSortList}(s, S_0), S) = \texttt{\#consSortList}(s, \texttt{\#appendSortList}(S_0, S))$$

TODO:: Start revising from here!!

The predicate symbol `#inSortList` checks list membership given a sort and a cons-list of sorts. It is defined as a symbol schema

$$\texttt{\#inXList}\{\#s\}\colon \texttt{\#X} \times \texttt{\#XList} \to \#s \quad \text{for any } \#s \in S_K$$

defines for each sort $\#s$ in the meta-theory K, a predicate symbol that checks list membership and returns either $\top_{\#s}$ or $\bot_{\#s}$:

$$\neg\texttt{\#inXList}\{\#s\}(x, \texttt{\#nilXList})$$
$$\texttt{\#inXList}\{\#s\}(x, \texttt{\#consXList}(y, L)) = (x = y) \vee \texttt{\#inXList}\{\#s\}(x, L)$$

Notice that using symbol (and axiom) schemas does not break the finitary of K as long as we do not use sort schemas. Sort schemas inductively define an infinite set of sorts, and thus will result in an infinite theory. While in K the number of sorts are designed to be finite (see Section 7.8 for a summary of all sorts, symbols, and axioms in K).

The functional symbol

$$\texttt{\#deleteXList}\colon \texttt{\#X} \times \texttt{\#XList} \to \texttt{\#XList}$$

takes an element and a list, and returns the list in which all the occurrences of the element have been deleted, and the order of the remaining elements does not change. It is axiomatized by the next two axioms

$$\texttt{\#deleteXList}(x, \texttt{\#nilXList}) = \texttt{\#nilXList}$$
$$\texttt{\#deleteXList}(x, \texttt{\#consXList}(y, L))$$
$$= ((x = y) \wedge \texttt{\#deleteXList}(x, L)) \vee ((x \neq y) \wedge \texttt{\#consXList}(y, \texttt{\#deleteXList}(x, L)))$$

Notation 6. To write list expressions more compactly, we use abbreviation

$$(x_1, x_2, \ldots, x_n) \equiv \texttt{\#consXList}(x_1, \texttt{\#consXList}(x_2, \ldots \texttt{\#consXList}(x_n, \texttt{\#nilXList}) \ldots))$$

7.5 Matching Logic Patterns

Matching Logic Variables. The sort `#Variable` is the sort for matching logic variables, with the only functional constructor

$$\texttt{\#variable}\colon \texttt{\#String} \times \texttt{\#Sort} \to \texttt{\#Variable}$$

in which the first argument is called the *name* of the variable, while the second argument is the sort of the variable.

Matching Logic Patterns. The sort `#Pattern` is the sort for matching logic patterns. It has in total five functional constructors, including the following four symbols

$$\texttt{\#application}\colon \texttt{\#Symbol} \times \texttt{\#PatternList} \to \texttt{\#Pattern}$$
$$\texttt{\#\and}\colon \texttt{\#Sort} \times \texttt{\#Pattern} \times \texttt{\#Pattern} \to \texttt{\#Pattern}$$
$$\texttt{\#\not}\colon \texttt{\#Sort} \times \texttt{\#Pattern} \to \texttt{\#Pattern}$$
$$\texttt{\#\exists}\colon \texttt{\#Sort} \times \texttt{\#Variable} \times \texttt{\#Pattern} \to \texttt{\#Pattern}$$

and an injection function from sort `#Variable` to sort `#Pattern`

$$\texttt{\#VariableToPattern}\colon \texttt{\#Variable} \to \texttt{\#Pattern}$$

Apart from the five constructor symbols, the sort `#Pattern` also has the following functional symbols for derived logic connectives

$$\texttt{\#\or}\colon \texttt{\#Sort} \times \texttt{\#Pattern} \times \texttt{\#Pattern} \to \texttt{\#Pattern}$$
$$\texttt{\#\implies}\colon \texttt{\#Sort} \times \texttt{\#Pattern} \times \texttt{\#Pattern} \to \texttt{\#Pattern}$$
$$\texttt{\#\iff}\colon \texttt{\#Sort} \times \texttt{\#Pattern} \times \texttt{\#Pattern} \to \texttt{\#Pattern}$$
$$\texttt{\#\forall}\colon \texttt{\#Sort} \times \texttt{\#Variable} \times \texttt{\#Pattern} \to \texttt{\#Pattern}$$
$$\texttt{\#\ceil}\colon \texttt{\#Sort} \times \texttt{\#Sort} \times \texttt{\#Pattern} \to \texttt{\#Pattern}$$
$$\texttt{\#\floor}\colon \texttt{\#Sort} \times \texttt{\#Sort} \times \texttt{\#Pattern} \to \texttt{\#Pattern}$$
$$\texttt{\#\equals}\colon \texttt{\#Sort} \times \texttt{\#Sort} \times \texttt{\#Pattern} \times \texttt{\#Pattern} \to \texttt{\#Pattern}$$
$$\texttt{\#\mem}\colon \texttt{\#Sort} \times \texttt{\#Sort} \times \texttt{\#Variable} \times \texttt{\#Pattern} \to \texttt{\#Pattern}$$
$$\texttt{\#\top}\colon \texttt{\#Sort} \to \texttt{\#Pattern}$$
$$\texttt{\#\bottom}\colon \texttt{\#Sort} \to \texttt{\#Pattern}$$

Notation 7. As a convention, we use φ and ψ for `#Pattern` variables, x, y, and z for `#String` variables, s for `#Sort` variables, v, u for `#Variable` variables, and σ for `#Symbol` variables.

Derived logic connectives are axiomatized by the following axioms

$$\texttt{\#\or}(s, \varphi, \psi) = \texttt{\#\not}(s, \texttt{\#\and}(s, \texttt{\#\not}(s, \varphi), \texttt{\#\not}(s, \psi)))$$
$$\texttt{\#\implies}(s, \varphi, \psi) = \texttt{\#\or}(s, \texttt{\#\not}(s, \varphi), \psi)$$
$$\texttt{\#\iff}(s, \varphi, \psi) = \texttt{\#\and}(s, \texttt{\#\implies}(s, \varphi, \psi), \texttt{\#\implies}(s, \psi, \varphi))$$
$$\texttt{\#\forall}(s, v, \varphi) = \texttt{\#\not}(s, \texttt{\#\exists}(s, v, \texttt{\#\not}(s, \varphi)))$$
$$\texttt{\#\ceil}(s_1, s_2, \varphi) = \texttt{\#application}(\texttt{\#`ceil}(s_1, s_2), \varphi)$$
$$\texttt{\#\floor}(s_1, s_2, \varphi) = \texttt{\#\not}(s_2, \texttt{\#\ceil}(s_1, s_2, \texttt{\#\not}(s_1, \varphi)))$$
$$\texttt{\#\equals}(s_1, s_2, \varphi, \psi) = \texttt{\#\floor}(s_1, s_2, \texttt{\#\iff}(s_1, \varphi, \psi))$$
$$\texttt{\#\mem}(s_1, s_2, v, \psi) = \texttt{\#\ceil}(s_1, s_2, \texttt{\#\and}(s_1, v, \psi))$$
$$\texttt{\#\top}(s) = \texttt{\#\exists}(s, \texttt{\#variable}(x, s), \texttt{\#variable}(x, s))$$
$$\texttt{\#\bottom}(s) = \texttt{\#\not}(s, \texttt{\#\top}(s))$$

Notation 8. As one may have already noticed, patterns of sort `#Pattern` get huge rather quickly. The following notation conventions are adopted to write `#Pattern` patterns in a more compact

way, by putting a bar upon their normal mixfix forms

$$\overline{x:s} \equiv \texttt{\#variable}(x, s)$$
$$\quad \text{or } \texttt{\#VariableToPattern}(\texttt{\#variable}(x, s))$$
$$\overline{\sigma(\varphi_1, \ldots, \varphi_n)} \equiv \texttt{\#application}(\sigma, (\varphi_1, \ldots, \varphi_n))$$
$$\overline{\varphi \wedge_s \psi} \equiv \texttt{\#\and}(s, \varphi, \psi)$$
$$\overline{\neg_s \varphi} \equiv \texttt{\#\not}(s, \varphi)$$
$$\overline{\exists_s v.\varphi} \equiv \texttt{\#\exists}(s, v, \varphi)$$
$$\overline{\varphi \vee_s \psi} \equiv \texttt{\#\or}(s, \varphi, \psi)$$
$$\overline{\varphi \rightarrow_s \psi} \equiv \texttt{\#\implies}(s, \varphi, \psi)$$
$$\overline{\varphi \leftrightarrow_s \psi} \equiv \texttt{\#\iff}(s, \varphi, \psi)$$
$$\overline{\forall_s v.\varphi} \equiv \texttt{\#\forall}(s, v, \varphi)$$
$$\overline{\lceil \varphi \rceil_{s_1}^{s_2}} \equiv \texttt{\#\ceil}(s_1, s_2, \varphi)$$
$$\overline{\lfloor \varphi \rfloor_{s_1}^{s_2}} \equiv \texttt{\#\floor}(s_1, s_2, \varphi)$$
$$\overline{\varphi =_{s_1}^{s_2} \psi} \equiv \texttt{\#\equals}(s_1, s_2, \varphi, \psi)$$
$$\overline{v \in_{s_1}^{s_2} \psi} \equiv \texttt{\#\mem}(s_1, s_2, v, \psi)$$
$$\overline{\top_s} \equiv \texttt{\#\top}(s)$$
$$\overline{\bot_s} \equiv \texttt{\#\bottom}(s)$$

Notice that we overload the notation $\overline{x:s}$ for both the `#Variable` pattern $\texttt{\#variable}(x, s)$ and the `#Pattern` pattern $\texttt{\#VariableToPattern}(\texttt{\#variable}(x, s))$, so that it can be used in both `#Variable` and `#Pattern` sort contexts, and as a result, it reduces the number of times we explicitly write the injection function `#VariableToPattern`.

Apart from the five constructors and ten derived connectives introduced above, the sort `#Pattern` also gets some common operators defined as functional symbols. Those functional symbols are defined and axiomatized in the following subsections.

Free Variable Collection. The functional symbol

$$\texttt{\#getFvs}\colon \texttt{\#Pattern} \rightarrow \texttt{\#VariableList}$$

traverses the argument pattern and collects all its free variables. If a variable has multiple occurrences in the pattern, it has the same number of occurrences in the result list. A related functional symbol is

$$\texttt{\#getFvsFromPatterns}\colon \texttt{\#PatternList} \rightarrow \texttt{\#VariableList}$$

which takes a list of patterns and applies `#getFvs` on each of them, and returns the concatenation of the results.

Notation 9. As a naming convention, we use L and R for `#PatternList` variables.

The next seven axioms define `#getFvs` and `#getFvsFromPatterns`, with the first five of them defining `#getFvs` and the last two defining `#getFvsFromPatterns`. Recall that we use (v) to denote

the singleton list that consists of only one element v (see Notation 6).

$$\begin{aligned}
&\texttt{\#getFvs}(v) = (v)\\
&\texttt{\#getFvs}(\overline{\sigma(L)}) = \texttt{\#getFvsFromPatterns}(L)\\
&\texttt{\#getFvs}(\overline{\varphi \wedge_s \psi}) = \texttt{\#appendPatternList}(\texttt{\#getFvs}(\varphi), \texttt{\#getFvs}(\psi))\\
&\texttt{\#getFvs}(\overline{\neg_s \varphi}) = \texttt{\#getFvs}(\varphi)\\
&\texttt{\#getFvs}(\overline{\exists_s v.\varphi}) = \texttt{\#deletePatternList}(v, \texttt{\#getFvs}(\varphi))\\
&\texttt{\#getFvsFromPatterns}(\texttt{\#nilPatternList}) = \texttt{\#nilPatternList}\\
&\texttt{\#getFvsFromPatterns}(\texttt{\#consPatternList}(\varphi, L))\\
&= \texttt{\#appendPatternList}(\texttt{\#getFvs}(\varphi), \texttt{\#getFvsFromPatterns}(L))
\end{aligned}$$

The predicate symbol schema

$$\texttt{\#occursFree}\{\#s\}\colon \texttt{\#Variable} \times \texttt{\#Pattern} \to \#s \quad \text{for any } \#s \in S_K$$

decides whether a variable occurs free in a given pattern

$$\texttt{\#occursFree}\{\#s\}(v, \varphi) = \texttt{\#inPatternList}\{\#s\}(v, \texttt{\#getFvs}(\varphi))$$

Fresh Variable Name Generation. The functional symbol

$$\texttt{\#freshName}\colon \texttt{\#PatternList} \to \texttt{\#String}$$

generates a fresh variable name that does not occur free in the argument list of patterns. It has the following axiom schema for any sort $\#s \in S_K$

$$\neg(\texttt{\#inPatternList}\{\#s\}(\texttt{\#variable}(\texttt{\#freshName}(L), s), \texttt{\#getFvsFromPatterns}(L)))$$

Substitution. The functional symbol

$$\texttt{\#substitute}\colon \texttt{\#Pattern} \times \texttt{\#Pattern} \times \texttt{\#Variable} \to \texttt{\#Pattern}$$

takes a target pattern φ, a "replace"-pattern ψ, and a "find"-variable v, and returns $\varphi[\psi/v]$, the pattern in which all free occurrences of variable v are placed with ψ with respect to alpha-renaming.

A related function is

$$\texttt{\#substitutePatterns}\colon \texttt{\#PatternList} \times \texttt{\#Pattern} \times \texttt{\#String} \times \texttt{\#Sort} \to \texttt{\#PatternList}$$

which takes a list of patterns and applies `#substitute` on each of them, and finally returns the list of all the results.

Notation 10. We abbreviate

$$\texttt{\#substitute}(\varphi, \psi, v) \equiv \overline{\varphi[\psi/v]}$$

and

$$\texttt{\#substitutePatterns}(L, \psi, v) \equiv \overline{L[\psi/v]}$$

for any `#PatternList` pattern L.

The functional symbols `#substitute` and `#substitutePatterns` have the following axioms. Recall that we use v, u for `#Variable` variables (see Notation 7), and L for `#PatternList` variable.

$$\overline{u[\psi/v]} = ((u = v) \wedge \psi) \vee ((u \neq v) \wedge u)$$
$$\overline{\sigma(L[\psi/v])} = \overline{\sigma(L[\psi/v])}$$
$$\overline{(\varphi_1 \wedge_s \varphi_2)[\psi/v]} = \overline{\varphi_1[\psi/v] \wedge_s \varphi_2[\psi/v]}$$
$$\overline{(\neg_s \varphi)[\psi/v]} = \overline{\neg_s \varphi[\psi/v]}$$
$$\overline{(\exists_s u.\varphi)[\psi/v]}$$
$$= \exists u'. \left(u' = \texttt{\#variable}(\texttt{\#freshName}(\varphi, \psi, v), \texttt{\#getSort}(u)) \wedge \overline{\exists_s u'.(\varphi[u'/u][\psi/v])}\right)$$
$$\overline{\texttt{\#nilPatternList}[\psi/v]} = \texttt{\#nilPatternList}$$
$$\overline{\texttt{\#consPatternList}(\varphi, L)[\psi/v]} = \texttt{\#consPatternList}(\overline{\varphi[\psi/v]}, \overline{L[\psi/v]})$$

Alpha-Renaming and Alpha-Equivalence. In matching logic, alpha-renaming is always assumed. This means that the pattern set of a matching logic theory is the one that is generated by the grammar of matching logic patterns in Figure 1, *modulo alpha-renaming*. In other words, matching logic patterns are equivalence classes with respect to alpha-equivalence. This fact is captured in K by the next axiom schema for any sort $\#s \in S_K$

$$\texttt{\#occursFree}\{\#s\}(v_1, \varphi) \wedge \texttt{\#occursFree}\{\#s\}(v_2, \varphi) \rightarrow \overline{\exists_s v_1.(\varphi[v_1/u])} = \overline{\exists_s v_2.(\varphi[v_2/u])}$$

7.6 Matching Logic Theories

We have defined a universe of meta-representations of matching logic sorts, symbols, and patterns, with which we represent all possible sorts, symbols, and patterns coming from all kinds of matching logic theories. When fix a theory T, the meta-representations of sorts, symbols, and patterns in the theory T only stand for a small portion of the universe of sort, symbol, and pattern meta-representations in K. For example, *Nat* is a sort in Presburger arithmetic, but not in lambda calculus. Pattern $f(x, y) = f(y, x)$ is an axiom in theories where f is a commutative symbol, but not an axiom in theories where f is not in the signature or not commutative.

Therefore, in the meta-theory K, we introduce three predicate schemas to capture what are defined in *the current theory* and what are not. They are:

$$\texttt{\#sortDeclared}\{\#s\}\colon \texttt{\#Sort} \rightarrow \#s$$
$$\texttt{\#symbolDeclared}\{\#s\}\colon \texttt{\#Symbol} \rightarrow \#s$$
$$\texttt{\#axiomDeclared}\{\#s\}\colon \texttt{\#Pattern} \rightarrow \#s$$

for any sort $\#s \in S_K$.

The definedness symbol $\lceil \_ \rceil_{s_1}^{s_2}$ is always declared if s_1 and s_2 are declared sorts:

$$\texttt{\#sortDeclared}\{\#s\}(s_1) \wedge \texttt{\#sortDeclared}\{\#s\}(s_2) \rightarrow \texttt{\#symbolDeclared}\{\#s\}(\texttt{\#`ceil}(s_1, s_2))$$

The symbol schema `#sortsDeclared` checks whether a list of sorts are declared:

$$\texttt{\#sortsDeclared}\{\#s\}\colon \texttt{\#SortList} \to \#s$$
$$\texttt{\#sortsDeclared}\{\#s\}(\texttt{\#nilSortList})$$
$$\texttt{\#sortsDeclared}\{\#s\}(\texttt{\#consSortList}(s, S))$$
$$= \texttt{\#sortDeclared}\{\#s\}(s) \wedge \texttt{\#sortsDeclared}\{\#s\}(S)$$

Wellformed Patterns. The predicate symbol schema

$$\texttt{\#wellFormed}\{s\}\colon \texttt{\#Pattern} \to s \quad \text{for any } s \in S_K$$

decides whether a pattern is *wellformed pattern*. A related predicate symbol `#wellFormedPatterns`$\{s\}$ checks whether multiple patterns are all wellformed

$$\texttt{\#wellFormedPatterns}\{s\}\colon \texttt{\#PatternList} \to s \quad \text{for any } s \in S_K$$

with two axioms

$$\texttt{\#wellFormedPatterns}\{s\}(\texttt{\#nilPatternList})$$
$$\texttt{\#wellFormedPatterns}\{s\}(\varphi, L) = \texttt{\#wellFormed}\{s\}(\varphi) \wedge \texttt{\#wellFormedPatterns}\{s\}(L)$$

The partial function

$$\texttt{\#getSort}\colon \texttt{\#Pattern} \rightharpoonup \texttt{\#Sort}$$

returns the sort of a pattern if the pattern is wellformed. Otherwise, it returns $\bot_{\texttt{\#Sort}}$. A related partial function is

$$\texttt{\#getSortsFromPatterns}\colon \texttt{\#PatternList} \rightharpoonup \texttt{\#SortList}$$

which takes a pattern lists and applies `#getSort` on each element, and returns the list of results. It has two straightforward axioms

$$\texttt{\#getSortsFromPatterns}(\texttt{\#nilPatternList}) = \texttt{\#nilSortList}$$
$$\texttt{\#getSortsFromPatterns}(\varphi, L) = \texttt{\#getSort}(\varphi), \texttt{\#getSortsFromPatterns}(L)$$

The partial functions `#wellFormed`$\{\#s\}$ and `#getSort` are defined by the following axioms

$$\texttt{\#wellFormed}\{\#s\}(\overline{x:s}) = \texttt{\#sortDeclared}\{\#s\}(s)$$
$$\texttt{\#wellFormed}\{\#s\}(\overline{\sigma(L)}) = \texttt{\#symbolDeclared}\{\#s\}(\sigma) \wedge \texttt{\#wellFormedPatterns}\{\#s\}(L)$$
$$\wedge (\texttt{\#getSortsFromPatterns}(L) = \texttt{\#getArgumentSorts}(\sigma))$$
$$\texttt{\#wellFormed}\{\#s\}(\overline{\varphi \wedge_s \psi}) = \texttt{\#wellFormed}\{\#s\}(\varphi) \wedge \texttt{\#wellFormed}\{\#s\}(\psi)$$
$$\wedge (\texttt{\#getSort}(\varphi) = s) \wedge (\texttt{\#getSort}(\psi) = s)$$
$$\texttt{\#wellFormed}\{\#s\}(\overline{\neg_s \varphi}) = \texttt{\#wellFormed}\{\#s\}(\varphi) \wedge (\texttt{\#getSort}(\varphi) = s)$$
$$\texttt{\#wellFormed}\{\#s\}(\overline{\exists_s v.\varphi}) = \texttt{\#wellFormed}\{\#s\}(v) \wedge \texttt{\#wellFormed}\{\#s\}(\varphi) \wedge (\texttt{\#getSort}(\varphi) = s)$$
$$\texttt{\#getSort}(\overline{x:s}) = \texttt{\#wellFormed}\{\#s\}(\overline{x:s}) \wedge s$$
$$\texttt{\#getSort}(\overline{\sigma(L)}) = \texttt{\#wellFormed}\{\#s\}(\overline{\sigma(L)}) \wedge \texttt{\#getReturnSort}(\sigma)$$
$$\texttt{\#getSort}(\overline{\varphi \wedge_s \psi}) = \texttt{\#wellFormed}\{\#s\}(\overline{\varphi \wedge_s \psi}) \wedge s$$
$$\texttt{\#getSort}(\overline{\neg_s \varphi}) = \texttt{\#wellFormed}\{\#s\}(\overline{\neg_s \varphi}) \wedge s$$
$$\texttt{\#getSort}(\overline{\exists_s v.\varphi}) = \texttt{\#wellFormed}\{\#s\}(\overline{\exists_s v.\varphi}) \wedge s$$

7.7 Matching Logic Proof System

The Hilbert-style proof system of matching logic in [16] defines the *derivability* of a pattern in an inductive manner. It defines a recursive set of patterns as *logic axioms* (in terms of logic schemas) and says all logic axioms are derivable. It also consists of four inference rules schemas—(Modus Ponens), (Universal Generalization), (Membership Introduction), (Membership Elimination)—that if some patterns (called the premises) are derivable, then a pattern (called the conclusion) is also derivable. The set of all derivable patterns according to the proof system is the smallest set of patterns that contains all logic axioms and are closed under the four inference rules schemas.

For the purpose of a unified perspective, we regard logic axiom schemas as inference rule schemas with zero premise.

Such inductiveness of derivability is naturally captured by introducing a predicate symbol

$$\texttt{\#provable}\{\#s\}\colon \texttt{\#Pattern} \to \#s \quad \text{for any } \#s \in S_K$$

which has axioms in correspondent to every inference rules:

$$\texttt{\#provable}\{\#s\}(\llbracket\varphi_1\rrbracket) \wedge \cdots \wedge \texttt{\#provable}\{\#s\}(\llbracket\varphi_n\rrbracket) \to \texttt{\#provable}\{\#s\}(\llbracket\psi\rrbracket)$$

is an axiom in the meta-theory if and only if

$$\frac{\varphi_1 \cdots \varphi_n}{\psi}$$

is an inference rule of matching logic, in which $\llbracket\varphi_1\rrbracket, \ldots, \llbracket\varphi_n\rrbracket, \llbracket\psi\rrbracket$ are representations of $\varphi_1, \ldots, \varphi_n, \psi$ in the meta-theory.

In the following, we list all the inference rules of the matching logic proof system and the correspondent axioms of `#provable`$\{\#s\}$. We recall the readers our conventions of choosing variable names. We use x and y for `#String` variables, s for `#Sort` variables, φ and ψ for `#Pattern` variables, L, R for `#PatternList` variables (in **Rule (M5)**). All axioms are parametric for any sort $\#s \in S_K$.

(Axiom). $A \vdash \varphi$ if $\varphi \in A$.

$$\texttt{\#axiomDeclared}\{\#s\}(\varphi) \to \texttt{\#provable}\{\#s\}(\varphi)$$

Propositional Logic Inference Rules.

(Propositional$_1$). $\vdash \varphi \to (\psi \to \varphi)$.

$$\forall\varphi\forall\psi\forall s\left(\texttt{\#wellFormed}\{\#s\}(\overline{\varphi \to_s (\psi \to_s \varphi)}) \to \texttt{\#provable}\{\#s\}(\overline{\varphi \to_s (\psi \to_s \varphi)})\right)$$

Just for simplicity, from now on we omit writing the wellformedness premises, though it does not mean they are unimportant. They are crucial in proving the faithfulness of the meta-theory. We also omit explicitly quantifying free variables.

(Propositional$_2$). $\vdash (\varphi_1 \to (\varphi_2 \to \varphi_3)) \to ((\varphi_1 \to \varphi_2) \to (\varphi_1 \to \varphi_3))$.

$$\texttt{\#provable}\{\#s\}(\overline{(\varphi_1 \to_s (\varphi_2 \to_s \varphi_3)) \to_s ((\varphi_1 \to_s \varphi_2) \to_s (\varphi_1 \to_s \varphi_3))}).$$

(Propositional$_3$). $\vdash (\neg\psi \to \neg\varphi) \to (\varphi \to \psi)$.

$$\texttt{\#provable}\{\#s\}(\overline{(\neg_s\psi \to_s \neg_s\varphi) \to_s (\varphi \to_s \psi)}).$$

(Modus Ponens). If $\vdash \varphi$ and $\vdash \varphi \to \psi$, then $\vdash \psi$.

$$\texttt{\#provable}\{\#s\}(\varphi) \wedge \texttt{\#provable}\{\#s\}(\overline{\varphi \to_s \psi}) \to \texttt{\#provable}\{\#s\}(\psi).$$

First-Order Logic With Equality Inference Rules.

($\forall$). $\vdash \forall v.(\varphi \to \psi) \to (\varphi \to \forall v.\psi)$ if v does not occur free in φ.

$$\neg\texttt{\#occursFree}(v, \varphi) \to \texttt{\#provable}\{\#s\}(\overline{\forall_s v.(\varphi \to_s \psi) \to_s (\varphi \to_s \forall_s v.\psi)}).$$

(Universal Generalization). If $\vdash \varphi$, then $\vdash \forall v.\varphi$.

$$\texttt{\#provable}\{\#s\}(\varphi) \to \texttt{\#provable}\{\#s\}(\overline{\forall_s v.\varphi}).$$

(Functional Substitution). $\vdash \exists u.u = \psi \wedge \forall v.\varphi \to \varphi[\psi/v]$ if u does not occur free in ψ.

$$\texttt{\#occursFree}(u, \psi) \to \texttt{\#provable}\{\#s\}(\overline{\exists_{s_2} u.u =^{s_2}_{s_1} \psi \wedge_{s_2} \forall_{s_2} v.\varphi \to_{s_2} \varphi[\psi/v]}).$$

(Functional Variable). $\vdash \exists u.u = v$

$$\texttt{\#provable}\{\#s\}(\overline{\exists_{s_2} u.u =^{s_2}_{s_1} v})$$

(Equality Introduction). $\vdash \varphi = \varphi$

$$\texttt{\#provable}\{\#s\}(\overline{\varphi =^{s_2}_{s_1} \varphi})$$

(Equality Elimination). $\vdash (\varphi_1 = \varphi_2) \to (\psi[\varphi_1/v] \to \psi[\varphi_2/v])$.

$$\texttt{\#provable}\{\#s\}(\overline{(\varphi_1 =^{s_2}_{s_1} \varphi_2) \to_{s_2} (\psi[\varphi_1/v] \to_{s_2} \psi[\varphi_2/v])}).$$

Definedness Axioms.

(Defined Variable). $\vdash \lceil x\!:\!s \rceil$.

$$\texttt{\#provable}\{\#s\}(\texttt{\#\textbackslash ceil}(s, s', \overline{x\!:\!s})).$$

Membership Rules.

(Membership Introduction). If $\vdash \varphi$, and v does not occur free in φ, then $\vdash v \in \varphi$.

$$\texttt{\#provable}\{\#s\}(\varphi) \wedge \neg\texttt{\#occursFree}\{\#s\}(v, \varphi) \to \texttt{\#provable}\{\#s\}(\overline{v \in^{s_2}_{s_1} \varphi}).$$

(Membership Introduction). If $\vdash v \in \varphi$ and v does not occur free in φ, then $\vdash \varphi$.

$$\texttt{\#provable}\{\#s\}(\overline{v \in^{s_2}_{s_1} \varphi}) \wedge \neg\texttt{\#occursFree}\{\#s\}(v, \varphi) \to \texttt{\#provable}\{\#s\}(\varphi).$$

(Membership Variable). $\vdash (v \in u) = (v = u)$.

$$\texttt{\#provable}\{\#s\}(\overline{(v \in^{s_2}_{s_1} u) =^{s_3}_{s_2} (v =^{s_2}_{s_1} u)}).$$

(Membership $\wedge$). $\vdash v \in (\varphi \wedge \psi) = (v \in \varphi) \wedge (v \in \psi)$.

$$\texttt{\#provable}\{\#s\}(\overline{v \in^{s_2}_{s_1} (\varphi \wedge_{s_1} \psi) =^{s_3}_{s_2} (v \in^{s_2}_{s_1} \varphi) \wedge_{s_2} (v \in^{s_2}_{s_1} \psi)}).$$

(Membership $\neg$). $\vdash v \in \neg\varphi = \neg(v \in \varphi)$.

$$\texttt{\#provable}\{\#s\}(\overline{v \in^{s_2}_{s_1} \neg_{s_1}\varphi =^{s_3}_{s_2} \neg_{s_2}(v \in^{s_2}_{s_1} \varphi)}).$$

(Membership $\forall$). $\vdash v \in \forall u.\varphi = \forall u.v \in \varphi$ if v is distinct from u.

$$(v \neq u) \rightarrow \texttt{\#provable}\{\#s\}(\overline{(v \in^{s_2}_{s_1} \forall_{s_1} u.\varphi) =^{s_3}_{s_2} (\forall_{s_2} u.(v \in^{s_2}_{s_1} \varphi))}).$$

(Membership Symbol). $\vdash v \in \sigma(\dots \varphi_i \dots) = \exists u.u \in \varphi_i \wedge v \in \sigma(\dots u \dots)$ where u is distinct from v and it does not occur free in $\sigma(\dots \varphi_i \dots)$.

$$\begin{aligned}&(u \neq v) \wedge \neg\texttt{\#occursFree}\{\#s\}(u, \overline{\sigma(L, \varphi_i, R)})\\ \rightarrow&\texttt{\#provable}\{\#s\}(\overline{v \in^{s_2}_{s_1} \sigma(L, \varphi_i, R) =^{s_3}_{s_2} \exists_{s_2} u.(u \in^{s_2}_{s_4} \varphi_i \wedge_{s_2} v \in^{s_2}_{s_1} \sigma(L, u, R))}),\end{aligned}$$

where L and R are `#PatternList` variables, and we write (L, φ, R) as a shorthand of

$$(L, \varphi, R) \equiv \texttt{\#appendPatternList}(L, \texttt{\#consPatternList}(\varphi, R)).$$

7.8 A Summary of the Meta-Theory

7.8.1 Sorts

Sort	Meaning
`#Char`	Characters
`#CharList` or `#String`	Finite lists of characters
`#Sort`	Matching logic sorts
`#SortList`	Finite lists of matching logic sorts
`#Symbol`	Matching logic symbols
`#SymbolList`	Finite lists of matching logic symbols
`#Variable`	Matching logic variables
`#VariableList`	Finite lists of matching logic variables
`#Pattern`	Matching logic patterns
`#PatternList`	Finite lists of matching logic patterns

7.8.2 Symbols

Symbols	Meaning
'a', 'b', ...	Individual characters
`#nilCharList` or `#epsilon`, `#consCharList`	Construct character lists (a.k.a. strings)
`#sort`	Construct sorts
`#symbol`	Construct symbols
`#getArgumentSorts`	Get the argument sorts of a symbol
`#getReturnSort`	Get the return sort of a symbol
`#‘ceil`	Definedness symbol(s)
`#nilSortList`, `#consSortList`	Construct sort lists
`#appendSortList`	Append two sort lists
`#inSortList`	Check whether a sort belongs to a sort list
`#deleteSortList`	Delete a sort from a sort list
`#variable`	Construct matching logic variables
`#application`	Construct symbol application patterns
`#\and`	Construct conjunction patterns
`#\not`	Construct negation patterns
`#\exists`	Construct existential quantification patterns
`#VariableToPattern`	Injection from variables to patterns
`#\or`	Construct disjunction patterns
`#\implies`,...	Construct all kinds of patterns
`#getFvs`	Get free variables in a pattern
`#getFvsFromPatterns`	Get all free variables from a pattern list
`#occursFree`	Check whether a variable occurs free in a pattern
`#freshName`	Generate a fresh variable name w.r.t. a pattern list
`#substitute`	Substitute a variable for a pattern
`#substitutePatterns`	Substitute a list of patterns
`#sortDeclared`	Check whether a sort is declared in the current theory
`#symbolDeclared`	Check whether a symbol is declared in the current theory
`#axiomDeclared`	Check whether an axiom is declared in the current theory
`#wellFormed`	Check whether a pattern is wellformed in the current theory
`#wellFormedPatterns`	Check whether a list of patterns are wellformed
`#getSort`	Get the sort of a pattern in the current theory
`#getSortsFromPatterns`	Get the list of sorts from a list of patterns
`#provable`	Check whether a pattern is provable in the current theory

8 Reflect Theories in the Meta-Theory

We have shown the readers the big picture of the meta-theory K and its usage in Section 7. In this section, we will show in detail how to *lift* any r.e. matching logic theory T to its meta-representation $[\![T]\!]$, and present the faithfulness theorem of K.

8.1 Naming Functions

A naming function associates any object to a string, called its *name*. For example, *Bool* is a non-parametric sort in Boolean algebra, and the string "Bool" is its name. *List* is a parametric sort, and the string "List" is its name. A naming function is a prerequisite for anything to be represented in the meta-theory K (see Section 7.2), and in Kore we assume the names of anything is itself wrapped with quotation marks.

Objects	**Names**
Non-parametric sort *Nat*	"Nat"
Parametric sort *List*	"List"
Non-parametric symbol *zero*	"zero"
Parametric symbol *nil*	"nil"
Parametric symbol *cons*	"cons"
Parametric symbol *append*	"append"

Table 1: Naming Function Example

In the following subsections, we fix a matching logic theory $T = (S, \Sigma, A)$ and a naming function *name*, and show how to get $[\![T]\!]$.

8.2 Reflect Sorts

The *sort lifting operation* is a mapping

$$[\![\_]\!] \colon S \to \textsc{Patterns}_{\texttt{\#Sort}}$$

inductively defined as

$$[\![s]\!] = \texttt{\#sort}(name(s), \texttt{\#nilSortList}) \qquad \text{if } s \in S \text{ is a basic sort}$$
$$[\![S\{s_1, \ldots, s_n\}]\!] = \texttt{\#sort}(name(S), [\![s_1, \ldots, s_n]\!]) \quad \text{if } S\{s_1, \ldots, s_n\} \in S \text{ is a sort schema instance}$$

where $[\![s_1, \ldots, s_n]\!]$ is the pointwise extension of sort lifting operation on a sequence of sorts:

$$[\![\_]\!] \colon S^* \to \textsc{Patterns}_{\texttt{\#SortList}}$$

inductively defined as

$$[\![\epsilon]\!] = \texttt{\#nilSortList}$$
$$[\![s]\!] = \texttt{\#consSortList}([\![s]\!], \texttt{\#nilSortList})$$
$$[\![s_1, \ldots, s_n]\!] = \texttt{\#consSortList}([\![s_1]\!], [\![s_2, \ldots, s_n]\!]) \quad \text{if } n \geq 2$$

If T has a basic sort $s \in S$, we add the following axiom to $[\![T]\!]$

$$\texttt{\#sortDeclared}\{\#s\}([\![s]\!])$$

If T has a sort schema

$$S\{s_1, \ldots, s_n\} \in S \quad \text{for any } s_1, \ldots, s_n \in S$$

we add the following axiom to $[\![T]\!]$

$$\texttt{\#sortDeclared}\{\#s\}(s_1) \wedge \cdots \wedge \texttt{\#sortDeclared}\{\#s\}(s_n) \to \texttt{\#sortDeclared}\{\#s\}([\![S\{s_1, \ldots, s_n\}]\!])$$

8.3 Reflect Symbols

The *symbol lifting operation* is a mapping

$$[\![\_]\!] \colon \Sigma \to \textsc{Patterns}_{\texttt{\#Symbol}}$$

For any basic symbol $\sigma \colon s_1 \times \cdots \times s_n \to s$, define

$$[\![\sigma]\!] = \texttt{\#symbol}(\mathit{name}(\sigma), \texttt{\#nilSortList}, [\![s_1, \ldots, s_n]\!], [\![s]\!])$$

and any symbol schema instance $\sigma\{s'_1, \ldots, s'_m\} \colon s_1 \times \cdots \times s_n \to s$, define

$$[\![\sigma\{s'_1, \ldots, s'_m\}]\!] = \texttt{\#symbol}(\mathit{name}(\sigma), [\![s'_1, \ldots, s'_m]\!], [\![s_1, \ldots, s_n]\!], [\![s]\!])$$

For T has a basic symbol σ, add the following axiom to $[\![T]\!]$

$$\texttt{\#symbolDeclared}\{\#s\}([\![\sigma]\!])$$

If T has symbol a symbol schema

$$\sigma\{s_1, \ldots, s_n\} \in S \quad \text{for any } s_1, \ldots, s_n \in S$$

then add the following axiom to $[\![T]\!]$

$$\texttt{\#sortDeclared}\{\#s\}(s_1) \wedge \cdots \wedge \texttt{\#sortDeclared}\{\#s\}(s_n) \to \texttt{\#symbolDeclared}\{\#s\}([\![\sigma\{s_1, \ldots, s_n\}]\!])$$

8.4 Reflect Variables

Define the set of variables in T

$$\textsc{Variables}_T = \bigcup_{x \in N \text{ is a variable name and } s \in S \text{ is a sort in } T} x \colon s$$

The *variable lifting operation*

$$[\![\_]\!]_0 \colon \textsc{Variables}_T \to \textsc{Patterns}_{\texttt{\#Variable}}$$

is defined as

$$[\![x \colon s]\!]_0 = \texttt{\#variable}(\mathit{name}(x), [\![s]\!])$$

8.5 Reflect Patterns and Axioms

Just like how we reflect sorts and symbols, patterns are reflected by lifting to their corresponding abstract syntax trees, too. Let us define the set of all patterns in T

$$\textsc{Patterns}_T = \bigcup_{s \in S \text{ is a sort in } T} \textsc{Patterns}_s$$

The *pattern lifting operation*

$$[\![\_]\!] \colon \textsc{Patterns}_T \to \textsc{Patterns}_{\texttt{\#Pattern}}$$

and its pointwise extension on pattern lists

$$[\![\_]\!]\colon \textsc{Patterns}^*_T \to \textsc{Patterns}_{\texttt{\#PatternList}}$$

are inductively defined as the follows.

$$\begin{aligned}
&[\![x\texttt{:}s]\!] = \texttt{\#VariableToPattern}([\![x\texttt{:}s]\!]_0)\\
&[\![\sigma(\varphi_1,\ldots,\varphi_n)]\!] = \texttt{\#application}([\![\sigma]\!],[\![\varphi_1,\ldots,\varphi_n]\!])\\
&[\![\varphi \wedge_s \psi]\!] = \texttt{\#\textbackslash and}([\![s]\!],[\![\varphi]\!],[\![\psi]\!])\\
&[\![\neg_s\varphi]\!] = \texttt{\#\textbackslash not}([\![s]\!],[\![\varphi]\!])\\
&[\![\exists^{s'}_s x\texttt{:}s.\varphi]\!] = \texttt{\#\textbackslash exists}([\![s]\!],[\![s']\!],[\![x\texttt{:}s]\!]_0,[\![\varphi]\!])\\
&[\![\varphi \vee_s \psi]\!] = \texttt{\#\textbackslash or}([\![s]\!],[\![\varphi]\!],[\![\psi]\!])\\
&[\![\varphi \to_s \psi]\!] = \texttt{\#\textbackslash implies}([\![s]\!],[\![\varphi]\!],[\![\psi]\!])\\
&[\![\varphi \leftrightarrow_s \psi]\!] = \texttt{\#\textbackslash iff}([\![s]\!],[\![\varphi]\!],[\![\psi]\!])\\
&[\![\forall^{s'}_s x\texttt{:}s.\varphi]\!] = \texttt{\#\textbackslash forall}([\![s]\!],[\![s']\!],[\![x\texttt{:}s]\!]_0,[\![\varphi]\!])\\
&[\![\lceil\varphi\rceil^{s'}_s]\!] = \texttt{\#\textbackslash ceil}([\![s]\!],[\![s']\!],[\![\varphi]\!])\\
&[\![\lfloor\varphi\rfloor^{s'}_s]\!] = \texttt{\#\textbackslash floor}([\![s]\!],[\![s']\!],[\![\varphi]\!])\\
&[\![\varphi =^{s'}_s \psi]\!] = \texttt{\#\textbackslash equals}([\![s]\!],[\![s']\!],[\![\varphi]\!],[\![\psi]\!])\\
&[\![x\texttt{:}s \in^{s'}_s \varphi]\!] = \texttt{\#\textbackslash mem}([\![s]\!],[\![s']\!],[\![x\texttt{:}s]\!]_0,[\![\varphi]\!])\\
&[\![\top_s]\!] = \texttt{\#\textbackslash top}([\![s]\!])\\
&[\![\bot_s]\!] = \texttt{\#\textbackslash bottom}([\![s]\!])
\end{aligned}$$

Notice that we do not desugar derivative connectives such as disjunction or universal quantification. Instead, we keep them as it is when lifting them to the meta-theory. Also notice how the injection wrapper `#VariableToPattern` is used.

If φ is an axiom in T, add the following axiom to $[\![T]\!]$

$$\texttt{\#axiomDeclared}\{\#s\}([\![\varphi]\!])$$

If φ is an axiom schema with $s_1,\ldots,s_n$ are sort parameters, add the following axiom to $[\![T]\!]$

$$\texttt{\#sortDeclared}\{\#s\}(s_1)\wedge\cdots\wedge\texttt{\#sortDeclared}\{\#s\}(s_n)\to\texttt{\#axiomDeclared}\{\#s\}([\![\varphi]\!])$$

8.6 Faithfulness

Theorem 11 (Faithfulness Theorem)**.** *Given a matching logic theory T and a naming function,*

$$T\vdash\varphi \quad \textit{iff} \quad K\cup[\![T]\!]\vdash\texttt{\#provable}\{\#s\}([\![\varphi]\!]) \quad \textit{for any } \#s\in S_K$$

Theorem 11 is an important theorem that justifies the design and usage of the theory K. It guarantees that the theory K does faithfully capture the matching logic reasoning in finite theories.

9 The Kore Language

The Kore language is a formal specification language that allows one to specify (both finite and recursively enumerable infinite) theories using a finite number of characters and space.

Before we present the syntax and semantics of Kore, we would like to recall the readers of upward and downward reflection relations that we have introduced in Section 7, a.k.a. the faithfulness theorem.

$$\frac{T \vdash \varphi}{K \cup [\![T]\!] \vdash_{\text{fin}} \texttt{\#provable}([\![\varphi]\!])} \text{(Upward Reflection)} \qquad \frac{K \cup [\![T]\!] \vdash_{\text{fin}} \texttt{\#provable}([\![\varphi]\!])}{T \vdash \varphi} \text{(Downward Reflection)}$$

Given such a faithful upward/downward reflection relation, it is suffice to specify the finite theory $[\![T]\!]$ in order to specify the possibly infinite theory T. However, the meta-theory K and the meta-representation $[\![T]\!]$ is inevitably more verbose and heavier than the original theory T.

The Kore language is one means to ease the discomfort of using the meta-theory. Kore provides a nice syntax surface that allows users to write declarations directly with the object-level theory T, while it still offers the full access to the meta-theory. Therefore, the users are free to write things at the object-level as they are used to, and are always able to write meta-level definitions and specifications to use the full power of the meta-theory and the faithful reflection relation. Object-level declarations and specifications are seen as just syntactic sugar of the more verbose meta-level declarations. This process of "desugaring object specifications to meta-specifications" are called *lifting*, which is giving Kore a semantics as a specification language (Section 9.2).

9.1 Kore Syntax

A Kore definition file is a sequence of ASCII characters. A Kore definition consists of a sequence of *Kore declarations*. In this section, we will completely define lexicon tokens and syntactic categories of the Kore language.

9.1.1 Comment in Kore

Kore supports C-style comments. Line comments start with "`//`" and block comments start with "`/*`" and end with "`*/`".

9.1.2 Lexicon Token

Kore has the following lexicon tokens. Some of them are reserved for future use.

Token Character(s)	Name	Usage
`:`	Colon	Variables and Declarations
`{ }`	Curly Braces	Sort Parameters
`( )`	Parenthesis	Kore Expressions
`,`	Comma	Kore Expressions
`#`	Sharp	Meta Identities
`\`	Slash	Reserved Identities
`'`	Prime	Identities
`::`	Double Colon	–
`.`	Period	–
`@`	At	–
`=`	Equals	–

9.1.3 Kore Keyword

The following keywords are used to initiate a Kore declaration.

Keyword	Usage
`sort`	Initiate a sort declaration
`symbol`	Initiate a symbol declaration
`axiom`	Initiate an axiom declaration

9.1.4 String Literal

Kore supports C-style string literals.

9.1.5 Word

Words are nonempty sequences of alphabets, digits, or the prime symbol "'". Words must start with an alphabet.

$$\langle \textit{word} \rangle ::= \texttt{[a-zA-Z][a-zA-Z0-9']}^*$$

9.1.6 Identifier

In Kore, an identifier is either a word, or a word followed after the sharp symbol "`#`" or the slash symbol "\", in which case we called the identifier *object*, *meta*, or *reserved*, respectively.

⟨*identifier*⟩ ::= ⟨*object-identifier*⟩ | ⟨*meta-identifier*⟩ | ⟨*reserved-identifier*⟩

⟨*object-identifier*⟩ ::= ⟨*word*⟩

⟨*meta-identifier*⟩ ::= '`#`' ⟨*word*⟩

⟨*reserved-identifier*⟩ ::= '\' ⟨*word*⟩

Identifiers are building blocks of many syntactic categories in Kore, such as ⟨*sort*⟩, ⟨*symbol*⟩, and ⟨*pattern*⟩. Notice that it is important to distinguish the three different categories of identifiers from one another, because Kore provides a nice unified syntactic interface that allows users to write

both matching logic connectives, object-level symbols, and meta-level symbols, in exactly the same prefix normal form, and we do not want to confuse the three of them. Instead, we use reserved identities for matching logic connectives, object identities for object-level symbols, and meta-level identities for meta-level symbols, as we will se in Section 9.1.13.

9.1.7 Sort

A sort is either a sort variable, or of the form $C_{sort}\{s_1, \ldots, s_n\}$ for some $n \geq 0$, in which C_{sort} is called a *sort constructor* and $s_1, \ldots, s_n$ is a comma-separated list of sorts. A sort is said to be an *object (or meta) sort*, if it is an object (or meta) sort variable, or it is constructed with an object (or meta) sort constructor and a list of object (or meta) sorts.

⟨*sort*⟩ ::= ⟨*object-sort*⟩ | ⟨*meta-sort*⟩

⟨*object-sort*⟩ ::= ⟨*object-sort-variable*⟩ | ⟨*object-sort-constructor*⟩ '`{`' ⟨*object-sort-list*⟩ '`}`'

⟨*meta-sort*⟩ ::= ⟨*meta-sort-variable*⟩ | ⟨*meta-sort-constructor*⟩ '`{`' ⟨*meta-sort-list*⟩ '`}`'

⟨*object-sort-variable*⟩ ::= ⟨*object-identifier*⟩

⟨*object-sort-constructor*⟩ ::= ⟨*object-identifier*⟩

⟨*meta-sort-variable*⟩ ::= ⟨*meta-identifier*⟩

⟨*meta-sort-constructor*⟩ ::= ⟨*meta-identifier*⟩

Here are some examples of object and meta-sorts written in Kore syntax.

Sort	Sort Constructor	Sort Parameters	Meta or Object?	Sort Schema?	Depending Variable(s)
`Nat{}`	`Nat`	`-`	Object	No	`-`
`List{Nat{}}`	`List`	`Nat{}`	Object	No	`-`
`Map{Nat{},S}`	`Map`	`Nat{},S`	Object	Yes	`S`
`Map{S1,List{S2}}`	`Map`	`S1,List{S2}`	Object	Yes	`S1,S2`
`S`	`-`	`-`	Object	Yes	`S`
`#Pattern{}`	`#Pattern`	`-`	Meta	No	`-`
`#Sort{}`	`#Sort`	`-`	Meta	No	`-`
`#PatternList{}`	`#PatternList`	`-`	Meta	No	`-`
`#S`	`-`	`-`	Meta	Yes	`#S`

We call a sort constructor C_{symbol} is a *parametric sort constructor*, if it requires at least one sort as its parameter(s). Otherwise, we call it a *non-parametric sort constructor*. The number of parameters that a sort constructor needs is called the *arity* of that sort constructor, which is decided in terms of a *sort declaration* that is discussed in Section 9.1.8.

Since the meta-theory is a finite theory, there is no parametric meta-sort constructor. All meta-sort constructors are non-parametric and have zero arity. Since meta-sort constructors are all non-parametric, meta-sort variables are not used to construct meta-sorts. Instead, they are used to construct meta-pattern schemas, which is discussed in Section 9.1.13.

The fixed number of predefined non-parametric meta-sort constructors in Kore are in correspondence to sorts in the meta-theory K (see Section 7.8).

9.1.8 Sort Declaration

An object sort declaration starts with "`sort`" keyword, followed by the object sort constructor being defined and a list of object sort variables wrapped with curly braces.

⟨*object-sort-declaration*⟩ ::= '`sort`' ⟨*object-sort-constructor*⟩ '`{`' ⟨*object-sort-variable-list*⟩ '`}`'

Here are some examples of sort declarations in Kore.

Declaration	Informal Semantics
`sort Nat{}`	Declare a non-parametric sort *Nat*
`sort List{S}`	Declare a sort schema $List\{s\}$ for any sort s
`sort Map{S1,S2}`	Declare a sort schema $Map\{s_1, s_2\}$ for any sort s_1, s_2

In Kore, new meta-sort constructors cannot be declared.

9.1.9 Symbol.

A symbol in Kore is of the form $C_{symbol}\{s_1, \ldots, s_n\}$ for some $n \geq 0$, in which C_{symbol} is called a *symbol constructor* and $s_1, \ldots, s_n$ is a comma-separated list of sorts. A symbol is said to be an object (or meta) symbol, if it is constructed with an object (or meta) symbol constructor and a list of object (or meta) sorts.

⟨*symbol*⟩ ::= ⟨*object-symbol*⟩ | ⟨*meta-symbol*⟩

⟨*object-symbol*⟩ ::= ⟨*object-symbol-constructor*⟩ '`{`' ⟨*object-sort-list*⟩ '`}`'

⟨*meta-symbol*⟩ ::= ⟨*meta-symbol-constructor*⟩ '`{`' ⟨*meta-sort-list*⟩ '`}`'

⟨*object-symbol-constructor*⟩ ::= ⟨*object-identifier*⟩

⟨*meta-symbol-constructor*⟩ ::= ⟨*meta-identifier*⟩

Here are some examples of object and meta-symbols written in Kore syntax.

Symbol	Symbol Constructor	Parameter(s)	Schema?	Depending Variable(s)
`zero{}`	`zero`	`-`	No	`-`
`succ{}`	`succ`	`-`	No	`-`
`nil{S}`	`nil`	`S`	Yes	`S`
`cons{S}`	`cons`	`S`	Yes	`S`
`cons{Nat}`	`cons`	`Nat`	No	`-`
`merge{S1,S2}`	`merge`	`S1,S2`	Yes	`S1,S2`
`merge{Nat,List{S}}`	`merge`	`Nat,List{S}`	Yes	`S`
`isLambdaTerm{#S}`	`isLambdaTerm`	`#S`	Yes	`#S`

We call a symbol constructor C_{symbol} a *parametric symbol*, if it requires at least one sort as sort parameter(s). Otherwise, we call it a *non-parametric symbol*. The number of parameters that a symbol constructor needs, together with its arity, argument sorts, and result sort, are defined in terms of symbol declarations, introduced in Section 9.1.10.

9.1.10 Symbol Declaration

Both object and meta-symbols can be declared with the keyword "`symbol`", followed by the symbol constructor being declared, a list of sort variables, a list of argument sorts, and a result sort.

⟨*symbol-declaration*⟩ ::= ⟨*object-symbol-declaration*⟩ | ⟨*meta-symbol-declaration*⟩

⟨*object-symbol-declaration*⟩ ::=
  | '`symbol`' ⟨*object-symbol-constructor*⟩ '`{`' ⟨*object-sort-variable-list*⟩ '`}`' '`(`' ⟨*object-sort-list*⟩ '`)`' '`:`' ⟨*object-sort*⟩

⟨*meta-symbol-declaration*⟩ ::=
  | '`symbol`' ⟨*meta-symbol-constructor*⟩ '`{`' ⟨*meta-sort-variable-list*⟩ '`}`' '`(`' ⟨*meta-sort-list*⟩ '`)`' '`:`' ⟨*meta-sort*⟩

Here are some examples of symbol declarations in Kore.

Declaration	Object or Meta?	Arity
`symbol zero{}():Nat{}`	Object	0
`symbol succ{}(Nat{}):Nat{}`	Object	1
`symbol nil{S}():List{S}`	Object	0
`symbol cons{S}(S,List{S}):List{S}`	Object	2
`symbol ceil{S1,S2}(S1):S2`	Object	1
`symbol isLambdaTerm{#S}(#Pattern{}):#S`	Meta	1

Notice that allowing parametric meta-symbols does not break the finitary of the meta-theory, because the number of meta-sorts in Kore is finite.

9.1.11 Logic Connective

In Kore, a *logic connective* (or simply a *connective*) is of the form $C_{connective}\{s_1, \ldots, s_n\}$ where $C_{connective}$ is a connective constructor and $s_1, \ldots, s_n$ is a list of sort parameters. A connective is said to be an object (or meta) connective, if it has a list of object (or meta) sort(s) as parameter(s). To distinguish from symbols, we use reserved identifiers for connective constructors.

⟨*connective*⟩ ::= ⟨*object-connective*⟩ | ⟨*meta-connective*⟩

⟨*object-connective*⟩ ::= ⟨*reserved-identifier*⟩ '`{`' ⟨*object-sort-list*⟩ '`}`'

⟨*meta-connective*⟩ ::= ⟨*reserved-identifier*⟩ '`{`' ⟨*meta-sort-list*⟩ '`}`'

The following are all logic connective constructors in Kore, including both basic connectives and derived connectives.

Logic Connective Constructor	Name	Logic Connective Constructor	Name
`\and`	Conjunction	`\not`	Negation
`\exists`	Existential Quantification	`\forall`	Universal Quantification
`\or`	Disjunction	`\equals`	Equality
`\mem`	Membership	`\implies`	Implication
`\iff`	Double Implication	`\top`	Top
`\bottom`	Bottom	`\ceil`	Ceiling
`\floor`	Flooring		

9.1.12 Head

Before we define the syntactic category ⟨*pattern*⟩ in Section 9.1.13, we would like to introduce a unified view of how *syntactically the same* that symbols and connectives (and aliases that we have not defined in this version of Kore) are. This unified view is useful to understand how patterns are constructed in Kore as well as the *lifting operation* when we define the semantics of Kore in Section 9.2.

A *head* is of the form $C\{s_1, \ldots, s_n\}$ in which C is called the *head constructor* and $s_1, \ldots, s_n$ are called the sort parameters. A head constructor is either a symbol constructor or a connective constructor. A head is said to be an object (or meta) head if it is an object (or meta) one as a symbol or a connective.

⟨*head*⟩ ::= ⟨*head-constructor*⟩ '`{`' ⟨*sort-list*⟩ '`}`'

⟨*head-constructor*⟩ ::= ⟨*symbol-constructor*⟩ | ⟨*connective-constructor*⟩

⟨*object-head*⟩ ::= ⟨*object-symbol*⟩ | ⟨*object-connective*⟩

⟨*meta-head*⟩ ::= ⟨*meta-symbol*⟩ | ⟨*meta-connective*⟩

9.1.13 Pattern

Apart from two exceptions: string literal patterns and variable patterns, patterns in Kore have a highly unified construction: they are constructed with a head and a list of patterns. Variable patterns are constructed with a *variable name* and a sort using the colon symbol "`:`". String literals are patterns on their own.

A pattern is said to be an object (or meta) pattern if it has an object (or meta) head, or it is a variable pattern with an object (or meta) variable name and an object (or meta) sort. String literals are meta-patterns.

⟨*pattern*⟩ ::= ⟨*object-pattern*⟩ | ⟨*meta-pattern*⟩

⟨*object-pattern*⟩ ::=
  | ⟨*object-head*⟩ '`(`' ⟨*pattern-list*⟩ '`)`'
  | ⟨*object-variable*⟩

⟨*meta-pattern*⟩ ::=
  | ⟨*meta-head*⟩ '`(`' ⟨*pattern-list*⟩ '`)`'
  | ⟨*meta-variable*⟩
  | ⟨*string-literal*⟩

⟨*object-variable*⟩ ::= ⟨*object-variable-name*⟩ '`:`' ⟨*object-sort*⟩

⟨*meta-variable*⟩ ::= ⟨*meta-variable-name*⟩ '`:`' ⟨*meta-sort*⟩

⟨*object-variable-name*⟩ ::= ⟨*object-identifier*⟩

⟨*meta-variable-name*⟩ ::= ⟨*meta-identifier*⟩

Notice that whether a pattern is an object or a meta-one depends only on its head, not the patter list that the head applies on.

9.1.14 Axiom Declarations

In Kore, a pattern can be declared as an axiom using the keyword "`axiom`", followed by a list of sort variables as parameters wrapped with curly braces.

⟨*axiom-declaration*⟩ ::= '`axiom`' '`{`' ⟨*sort-variable-list*⟩ '`}`' ⟨*pattern*⟩

If the pattern is an object pattern, we call the declaration an *object axiom declaration*. Otherwise, it is a *meta-axiom declaration*.

9.1.15 Wellformedness

9.2 Kore Semantics

Give a theory T defined as a Kore definition "`definition.kore`" which consists of a set of object sort declarations, a set of object and meta-symbol declarations, and a set of object and meta-axiom declarations using the syntax that we have introduced before. The *semantics* of the theory T, known as *the meta-theory instantiated with* T, is given by *lifting* the Kore definition "`definition.kore`" to the Kore definition "`#definition.kore`" which encodes $[\![T]\!]$. The Kore definition "`#definition.kore`" will contain only meta-symbol declarations and meta-axiom declarations. The lifting algorithm is defined in the next few sections.

9.2.1 General Principle of Lifting

The general principle of the lifting operation is to lift whatever object level targets to their corresponding meta-level representations. An object sort/symbol/pattern is lifted to the meta-pattern that is the abstract syntax tree of it. An object sort/symbol/axiom declaration is lifted to the meta-level axiom declaration that declares "the object sort/symbol/axiom is declared".

Here is one more word about lifting a pattern. Recall that most patterns are constructed with a head and a list of patterns, and the list of patterns can be a mixture of object and meta-patterns. If the pattern-being-lifted is an object pattern, both its head and list of subpatterns are lifted. If the pattern-being-lifted is a meta-pattern, only its subpatterns are lifted, and its head is not lifted.

Notation 12. We use the double bracket $[\![\_]\!]$ for lifting operation and overload it for all syntactic categories. Subscripts are used (e.g., $[\![\_]\!]_{name}$, $[\![\_]\!]_{list}$, ...) when we need to define some auxiliary lifting operations and do not want to confuse them with the canonical one $[\![\_]\!]$.

9.2.2 Lift Object Identifiers To String Literals

Object identifiers are lifted to string literals by wrapping them with quotation marks.

$$[\![\_]\!]_{name} : \langle \textit{object-identifier} \rangle \to \langle \textit{string-literal} \rangle$$
$$[\![x]\!]_{name} = \text{`"'}\ x\ \text{`"'} \quad \text{for an object identifier } x$$

9.2.3 Lift Object & Reserved Identifiers To Meta Identifiers

Object and reserved identifiers are lifted to meta-identifiers by appending the sharp symbol '`#`' in front.

$$[\![\_]\!]_\sharp : \langle \textit{object-identifier} \rangle \cup \langle \textit{reserved-identifier} \rangle \to \langle \textit{meta-identifier} \rangle$$
$$[\![x]\!]_\sharp = \text{`\#'}\ x \quad \text{for any object or reserved identifier } x$$

9.2.4 Lift Object Sort Constructors To Meta Symbols

Object sort constructors are lifted to non-parametric meta-symbols as

$$\begin{aligned}&[\![\_]\!]\colon \langle \textit{object-sort-constructor}\rangle \to \langle \textit{meta-symbol}\rangle\\ &[\![C_{sort}]\!] = [\![C_{sort}]\!]_\sharp\{\} \quad \text{for any object sort constructor } C_{sort}\end{aligned}$$

For example,

Object Sort Constructor C_{sort}	Non-Parametric Meta Symbol $[\![C_{sort}]\!]$
`Nat`	`#Nat{}`
`List`	`#List{}`
`Map`	`#Map{}`

9.2.5 Lift Object Sorts To Meta-Patterns

Object sorts are lifted to meta-patterns by the lifting operation inductively defined as

$$\begin{aligned}&[\![\_]\!]\colon \langle \textit{object-sort}\rangle \to \langle \textit{meta-pattern}\rangle\\ &[\![s]\!] = \begin{cases} [\![S]\!]_\sharp \text{ `:' `\#Sort'} & \text{if } s \text{ is an object sort variable } S\\ [\![C_{sort}]\!]\texttt{(}[\![s_1]\!]\texttt{,...,}[\![s_n]\!]\texttt{)} & \text{if } s \text{ is } C_{sort}\{s_1,\ldots,s_n\}\\ & \text{where } C_{sort} \text{ is an object sort constructor}\\ & \text{and } s_1,\ldots,s_n \text{ are object sorts}\end{cases}\end{aligned}$$

For example,

Object Sort s	Meta Pattern $[\![s]\!]$
`Nat{}`	`#Nat{}()`
`List{Nat{}}`	`#List{}(#Nat{}())`
`Map{Nat{},S}`	`#Map{}(#Nat{}(),#S:#Sort)`
`Map{S1,List{S2}}`	`#Map{}(#S1:#Sort,#List{}(#S2:#Sort))`
`S`	`#S:#Sort`

9.2.6 Lift Object Sort Lists To Meta-Patterns

Given a list of sorts $s_1,\ldots,s_n$. We can naturally define the element-wise extension lifting operation as

$$\begin{aligned}&[\![\_]\!]\colon \langle \textit{object-sort-list}\rangle \to \langle \textit{meta-pattern-list}\rangle\\ &[\![s_1,\ldots,s_n]\!] = [\![s_1]\!],\ldots,[\![s_n]\!]\end{aligned}$$

However, it is often the case that we want to lift a sort list to *one* `#SortList` *meta-pattern* instead of *a list of* `#Sort` *meta-patterns*. For that reason, we define the following *list-lifting operation* for sorts:

$$\begin{aligned}&[\![\_]\!]_{list}\colon \langle \textit{object-sort-list}\rangle \to \langle \textit{meta-pattern}\rangle\\ &[\![\alpha]\!] = \begin{cases} \texttt{\#nilSortList} & \text{if } \alpha \text{ is empty}\\ \texttt{\#consSortList(}[\![s]\!], [\![\beta]\!]_{list}\texttt{)} & \text{if } \alpha = s,\beta\end{cases}\end{aligned}$$

9.2.7 Lift Sort Declarations To Symbol & Axiom Declarations

A sort declaration is lifted to a symbol declaration and two axiom declarations

$$[\![\_]\!]\colon \langle \textit{sort-declaration} \rangle \to \langle \textit{declaration-list} \rangle$$

defined as follows. For any sort schema $C_{sort}\{S_1, \ldots, S_n\}$ with sort constructor C_{sort} and sort variables $S_1, \ldots, S_n$, its lifting is

$$\begin{aligned}
&[\![\texttt{sort } C_{sort}\{S_1\texttt{,}\ldots\texttt{,}S_n\}]\!] \\
=\ &\texttt{symbol } [\![C_{sort}]\!]\underbrace{\texttt{(\#Sort\{\},}\ldots\texttt{,\#Sort\{\})}}_{n \text{ times}}\texttt{:\#Sort\{\}} \\
&\texttt{axiom\{\#S\} \textbackslash equals\{\#Sort\{\},\#S\}(} \\
&\qquad\qquad [\![C_{sort}\{S_1, \ldots, S_n\}]\!]\texttt{,} \\
&\qquad\qquad \texttt{\#sort\{\}(}[\![C_{sort}]\!]_{name}\texttt{,}\ [\![S_1, \ldots, S_m]\!]_{list}\texttt{))} \\
&\texttt{axiom\{\#S\} \textbackslash implies\{\#S\}(} \\
&\qquad\qquad \texttt{\#sortsDeclared\{\#S\}(}[\![S_1, \ldots, S_m]\!]_{list}\texttt{),} \\
&\qquad\qquad \texttt{\#sortDeclared\{\#S\}(}[\![C_{sort}\{S_1, \ldots, S_n\}]\!]\texttt{))}
\end{aligned}$$

9.2.8 Lift Object Symbol Constructors To Meta Symbols

Object symbol constructors are lifted to non-parametric meta-symbols as

$$\begin{aligned}
&[\![_]\!]\colon \langle \textit{object-symbol-constructor} \rangle \to \langle \textit{meta-symbol} \rangle \\
&[\![C_{symbol}]\!] = [\![C_{symbol}]\!]_\sharp\{\} \quad \text{for any object symbol constructor } C_{symbol}
\end{aligned}$$

9.2.9 Lift Object Symbol Declarations To Symbol & Axiom Declarations

An object symbol declaration

$$\texttt{symbol } C_{symbol}\{S_1\texttt{,}\ldots\texttt{,}S_m\}\texttt{(}s_1\texttt{,}\ldots\texttt{,}s_n\texttt{):}s$$

with object symbol constructor C_{symbol}, sort variables $S_1, \ldots, S_m$, argument sorts $s_1, \ldots, s_n$, and return sort s, is lifted as

$$\begin{aligned}
&\texttt{symbol } C_{symbol}\{S_1\texttt{,}\ldots\texttt{,}S_m\}\texttt{(}s_1\texttt{,}\ldots\texttt{,}s_n\texttt{):}s \\
=\ &\texttt{symbol } [\![C_{symbol}]\!]\underbrace{\texttt{(\#Sort\{\},}\ldots\texttt{,\#Sort\{\})}}_{m \text{ times}}\texttt{,}\underbrace{\texttt{\#Pattern\{\},}\ldots\texttt{,\#Pattern\{\})}}_{n \text{ times}}\texttt{:\#Pattern\{\}} \\
&\texttt{axiom\{\#S\} \textbackslash equals\{\#Pattern\{\},\#S\}(} \\
&\qquad\qquad [\![C_{symbol}]\!]\texttt{(}[\![S_1]\!]\texttt{,}\ldots\texttt{,}[\![S_m]\!]\texttt{,\#P1:\#Pattern\{\},}\ldots\texttt{,\#Pn:\#Pattern\{\}),} \\
&\qquad\qquad \texttt{\#application\{\}(}[\![C_{symbol}]\!]_{symbol}\texttt{,}\ collapse_{\texttt{\#Pattern}}\texttt{(\#P1:\#Pattern\{\},}\ldots\texttt{,\#Pn:\#Pattern\{\})))} \\
&\texttt{axiom\{\#S\} \#\textbackslash implies\{\#S\}(}[\![S_1, \ldots, S_n]\!]_{declared}\texttt{, \#symbolDeclared\{\#S\}(}[\![C_{symbol}]\!]_{symbol}\texttt{))}
\end{aligned}$$

in which $[\![\_]\!]_{symbol}$ (given $S_1, \ldots, S_m, s_1, \ldots, s_n, s$, and `#S`) is defined as

$$[\![C_{symbol}]\!]_{symbol} = \texttt{\#symbol(}[\![C_{symbol}]\!]_{name}\texttt{,}[\![S_1, \ldots, S_m]\!]_{list}\texttt{,}[\![s_1, \ldots, s_n]\!]_{list}\texttt{,}[\![s]\!]\texttt{))}$$

and $collapse_{\texttt{\#Pattern}}(\alpha)$ for a list α of Kore patterns is defined as

$$collapse_{\texttt{\#Pattern}}(\alpha) = \begin{cases} \texttt{\#nilPatternList}\{\} & \text{if } \alpha \text{ is empty list} \\ \texttt{\#consPatternList}\{\}(\varphi, collapse_{\texttt{\#Pattern}}(\beta)) & \text{if } \alpha \text{ is } \varphi, \beta \end{cases}$$

9.2.10 Lift Connective Constructors To Meta Symbols

Connective constructors are lifted to non-parametric meta-symbols as follows.

$$\begin{aligned} &[\![\_]\!] \colon \langle \textit{connective} \rangle \to \langle \textit{meta-symbol} \rangle \\ &[\![C_{logic}]\!] = [\![C_{logic}]\!]_\sharp\{\} \quad \text{for any connective constructor } C_{connective} \end{aligned}$$

9.2.11 Lift Object Head Constructors To Meta Symbols

Recall that a head constructor is either a symbol constructor or a connective constructor. A head constructor C is lifted to the meta-symbol $[\![C]\!]$ in the same way a symbol/connective constructor is lifted.

9.2.12 Lift Object Variables To Meta Patterns of Sort `#Variable`

Variables are lifted to meta-patterns of sort `#Variable` as follows.

$$\begin{aligned} &[\![\_]\!]_0 \colon \langle \textit{object-variable} \rangle \to \langle \textit{meta-pattern} \rangle \\ &[\![x : s]\!] = \texttt{\#variable}\{\}(\texttt{"}[\![x]\!]_{name}\texttt{"},[\![s]\!]) \quad \text{for any object variable name } x \text{ and object sort } s \end{aligned}$$

9.2.13 Lift Patterns To Meta Patterns of Sort `#Pattern`

One of the main part of the lifting operation is lifting patterns to meta- patterns of sort `#Pattern`, which is the main job of this subsection. We recall the readers of the following grammar of Kore patterns, in which we intensionally merge the grammars of both object and meta-patterns for simplicity.

$\langle \textit{pattern} \rangle$ ::=
- $\langle \textit{object-identifier} \rangle$ ':' $\langle \textit{object-sort} \rangle$
- $\langle \textit{meta-identifier} \rangle$ ':' $\langle \textit{meta-sort} \rangle$
- $\langle \textit{string-literal} \rangle$
- $\langle \textit{object-head} \rangle$ '(' $\langle \textit{pattern-list} \rangle$ ')'
- $\langle \textit{meta-head} \rangle$ '(' $\langle \textit{pattern-list} \rangle$ ')'

We will define the lifting operation for patterns case by case following the grammar of patterns

$$[\![\_]\!] \colon \langle \textit{pattern} \rangle \to \langle \textit{pattern} \rangle$$

Case A: Object Variable Pattern.

$$[\![v]\!] = \texttt{\#VariableToPattern}([\![v]\!]_0) \quad \text{if } v \text{ is an object variable}$$

Case B: Meta Variable Pattern.

$$[\![v]\!] = v \quad \text{if } v \text{ is a meta-variable}$$

Case C: String Literal.

$$[\![s]\!] = s \quad \text{if } s \text{ is a string literal}$$

Case D & E: Headed Pattern. A bit extra work is need to lift headed patterns. This is because we have defined two ways to lift an object variable. One way is to lift it to a `#Variable` pattern using the lifting operation $[\![\_]\!]_0$. The other way is to lift it to a `#Pattern` using the lifting operation $[\![\_]\!]$, which we just defined the case for object variables in Case A.

In practice, some heads (for example `\exists` and `\forall`) expect some of its argument to be a variable instead of an arbitrary pattern, and we want to reflect that fact in the lifting and the meta-theory. In Section 9.1.15 we have defined for all head constructors C a *variable-pattern vector* ν_C. The vector is of the same length as the arity of C, and each of its component is from a set of two flags: $\{\mathtt{v}, \mathtt{p}\}$. We use ν_C^i to denote the ith component of the vector ν_C. The lifting operation replies on such variable-pattern vectors.

Given any headed pattern $C\{s_1, \ldots, s_n\}(\varphi_1, \ldots, \varphi_n)$, its variable-pattern vector ν_C, and one sub-pattern φ_i, define the customized lifting for $[\![\varphi_i]\!]_{vp}$ as follows

$$[\![\varphi_i]\!]_{vp} = \begin{cases} [\![\varphi_i]\!] & \text{if } \nu_C^i = \mathtt{p} \\ [\![\varphi_i]\!]_0 & \text{if } \nu_C^i = \mathtt{v} \end{cases}$$

The lifting for headed patterns are therefore defined as follows.

$$[\![C\{s_1, \ldots, s_n\}(\varphi_1, \ldots, \varphi_n)]\!] = \begin{cases} [\![C]\!]([\![s_1]\!], \ldots, [\![s_n]\!], [\![\varphi_1]\!]_{vp}, \ldots, [\![\varphi_n]\!]_{vp}) \\ C\{s_1, \ldots, s_n\}([\![\varphi_1]\!]_{vp}, \ldots, [\![\varphi_n]\!]_{vp}) \end{cases}$$

9.2.14 Lift Axiom Declarations To Axiom Declarations

In Kore, an axiom declaration is lifted as follows.

$$[\![\_]\!] \colon \langle \textit{axiom-declaration} \rangle \to \langle \textit{axiom-declaration} \rangle$$

$$[\![\mathtt{axiom}\{s_1, \ldots, s_n\}\ \ \varphi]\!] = \begin{cases} \mathtt{axiom}\{s_1', \ldots, s_m'\}\ \ \mathtt{\#provable}([\![\varphi]\!]) & \text{if } \varphi \text{ is an object pattern} \\ \mathtt{axiom}\{s_1', \ldots, s_m'\}\ \ [\![\varphi]\!] & \text{if } \varphi \text{ is a meta-pattern} \end{cases}$$

where $s_1', \ldots, s_m'$ are all those meta-variables in $s_1, \ldots, s_n$.

9.3 Example A: Presburger Arithmetic

```
1  sort Nat{}
2
3  symbol zero{}() : Nat{}
4  symbol succ{}(Nat{}) : Nat{}
5  symbol plus{}(Nat{}, Nat[}) : Nat{}
```

9.4 Example B: Parametric Lists

```
sort List{S}

symbol nil{S}() : List{S}
symbol cons{S}(S, List{S}) : List{S}
symbol append{S}(List{S}, List{S}) : List{S}
symbol rev{S}(List{S}) : List{S}

axiom{S,S'}
  \equals{List{S},S'}(
     append{S}(nil{S}(), L:List{S}),
     L:List{S})

axiom{S,S'}
  \equals{List{S},S'}(
     append{S}(cons{S}(X:S, L:List{S}), L':List{S}),
     cons{S}(X:S, append{S}(L:List{S}, L':List{S})))
```

9.5 Example C: Lambda Calculus

```
sort Exp{}

symbol app{}(Exp{}, Exp{}) : Exp{}
symbol lambda0{}(Exp{}, Exp{}) : Exp{}

/* Define the alias lambda directly at the meta-level */
symbol #lambda{}(#Variable{}, #Pattern{}) : #Pattern{}
axiom{#S}
  \equals{#Pattern{},#S}(
    #lambda{}(#V:#Variable{}, #E:#Pattern{}),
    \exists{Exp,Exp}(
      #V:#Variable{},
      lambda0{}(
        #variableAsPattern{}(#V:#Variable{}),
        #E:#Pattern{})))

/* Define isLambdaTerm for each (meta-level) result sort #S */
symbol #isLambdaTerm{#S}(#Pattern{}) : #S
axiom{#S}
  #isLambdaTerm{#S}(
    #variableAsPattern{}(
      #variable{}(X:#String{}, #Exp{}())))
axiom{#S}
  \equals{#S,#S}(
    #isLambdaTerm{#S}(
      app{}(#E:#Pattern{}, #E':#Pattern{})),
    \and{#S}(
       #isLambdaTerm{#S}(#E:#Pattern{}),
       #isLambdaTerm{#S}(#E':#Pattern{}))))
axiom{#S}
  \equals{#S,#S}(
    #isLambdaTerm{#S}(#lambda{}(#V:#Variable{}, #E:#Pattern{})),
    \and{#S}(
      #variableHasSort{#S}(#V:#Variable, #Exp{}()),
      #isLambdaTerm{#S}(#E:#Pattern)))
```

```
36
37  /* Define the beta reduction axiom:
38   * app(lambda(x,e),e') = e'[e/x]
39   * if x is an Exp variable and e and e' are lambda terms
40   */
41  axiom{#S,R}
42    \implies{#S}(
43      \and{#S}(\and{#S}(
44        #variableHasSort{#S}(#V:#Variable{}, #Exp{}()),
45        #isLambdaTerm{#S}(#E:#Pattern{})),
46        #isLambdaTerm{#S}(#E':#Pattern{})),
47      #provable{#S}(
48        \equals{Exp{},R}(
49          app{}(
50            #lambda{}(#V:#Variable{}, #E:#Pattern{}),
51            #E':#Pattern{}),
52          #substitute{}(
53            #E':#Pattern{},
54            #E:#Pattern{},
55            #V:#Variable{})))))
```

9.6 Example D: Contexts

9.7 Get Things From Here

9.7.1 Formal BNF Grammar of Kore

⟨*definition*⟩ ::= ⟨*declarations*⟩

⟨*declarations*⟩ ::= '' | ⟨*declaration*⟩ | ⟨*declaration*⟩ ⟨*declarations*⟩

⟨*declaration*⟩ ::=
  | ⟨*sort-declaration*⟩
  | ⟨*symbol-declaration*⟩
  | ⟨*axiom-declaration*⟩
  | ⟨*alias-declaration*⟩

⟨*sort-declaration*⟩ ::=
  | '`sort`' ⟨*sort*⟩
  | '`hooked-sort`' ⟨*sort*⟩

⟨*sort*⟩ ::=
  | ⟨*atomic-sort*⟩
  | ⟨*parametric-sort*⟩

⟨*parametric-sort*⟩ ::=
  | ⟨*parametric-sort-constructor*⟩ '`{`' ⟨*sort-list*⟩ '`}`'

⟨*sort-list*⟩ ::=
  | ⟨*sort*⟩
  | ⟨*sort*⟩ '`,`' ⟨*sort-list*⟩

⟨*atomic-sort*⟩ ::= `[A-Z][a-zA-Z0-9]`$^{+}$

⟨*sort-variable*⟩ ::= `[A-Z0-9]`$^+$

⟨*parametric-sort-constructor*⟩ ::= `[A-Z][a-zA-Z0-9]`$^+$

⟨*symbol-declaration*⟩ ::=
  | '`symbol`' ⟨*symbol*⟩ ⟨*symbol-signature*⟩
  | '`hooked-symbol`' ⟨*symbol*⟩ ⟨*symbol-signature*⟩

⟨*symbol*⟩ ::=
  | ⟨*nonparametric-symbol*⟩
  | ⟨*parametric-symbol*⟩

⟨*nonparametric-symbol*⟩ ::= `[a-zA-Z0-9]`$^+$

⟨*parametric-symbol*⟩ ::=
  | ⟨*parametric-symbol-name*⟩ '`{`' ⟨*sort-list*⟩ '`}`'

⟨*parametric-symbol-name*⟩ ::= `[a-zA-Z0-9]`$^+$

⟨*symbol-signature*⟩ ::=
  | '`(`' ⟨*sort-list*⟩ '`)`' '`:`' ⟨*sort*⟩

⟨*axiom-declaration*⟩ ::=
  | '`axiom`' ⟨*pattern*⟩

⟨*pattern*⟩ ::=
  | ⟨*variable-name*⟩ '`:`' ⟨*sort*⟩
  | ⟨*symbol*⟩ '`(`' ⟨*pattern-list*⟩ '`)`'
  | '`\and`' '`(`' ⟨*pattern*⟩ '`,`' ⟨*pattern*⟩ '`,`' ⟨*sort*⟩ '`)`'
  | '`\not`' '`(`' ⟨*pattern*⟩ '`,`' ⟨*sort*⟩ '`)`'
  | '`\exists`' '`(`' ⟨*variable-name*⟩ '`:`' ⟨*sort*⟩ '`,`' ⟨*pattern*⟩ '`,`' ⟨*sort*⟩ '`)`'
  | `how to denote meta-variables?`

⟨*pattern-list*⟩ ::=
  | ''
  | ⟨*pattern*⟩
  | ⟨*pattern*⟩ '`,`' ⟨*pattern-list*⟩

⟨*alias-declaration*⟩ ::=
  | '`alias`' `todo here, check the following example of IMP.`

10 Proof of Faithfulness Theorem

Before we start to prove the theorem, we need to explain what "the corresponding patterns" $\hat{T}$ and $\hat{\varphi}$ are, as they appear in the theorem. For that reason, we introduce the next definition.

Definition 13 (Naming and Lifting)**.** Suppose $T = (S, \Sigma, A)$ is a finite matching logic theory, with $Var = \bigcup_{s \in S} Var_s$ is the set of all variables of T. *A naming of* T, denoted as e, consists of the following three naming functions

- A sort-naming function $e_S \colon S \to \textsc{Patterns}_{\texttt{\#String}}$ that maps each sort in S to a syntactic `#String` pattern such that $K \vdash e_S(s_1) \neq e_S(s_2)$ for any distinct sorts s_1 and s_2;

- A symbol-naming function $e_\Sigma \colon \Sigma \to \textsc{Patterns}_{\texttt{\#String}}$ that maps each symbol in Σ to a syntactic `#String` pattern such that $K \vdash e_\Sigma(\sigma_1) \neq e_\Sigma(\sigma_2)$ for any distinct symbols σ_1 and σ_2;
- A variable-naming function $e_{Var} \colon Var \to \textsc{Patterns}_{\texttt{\#String}}$ that maps each variable in T to a syntactic `#String` pattern in K, such that $K \vdash e_{Var}(x) \neq e_{Var}(y)$ for any distinct variables x and y.

Given $e = \{e_S, e_\Sigma, e_{Var}\}$ is a naming of theory T, the *lift of T with respect to e* consists of the following lifting functions.

(Sort-lifting). For each sort s in theory T, the lift of s is a `#Sort{}` pattern

$$\hat{s} = \texttt{\#sort}(e_S(s))$$

(Symbol-lifting). For each symbol $\sigma \in \Sigma_{s_1 \ldots s_n, s}$, the lift of σ is a `#Symbol` pattern

$$\hat{\sigma} = \texttt{\#symbol}(e_\Sigma(\sigma), (\hat{s_1}, \ldots, \hat{s_n}), \hat{s})$$

(Pattern-lifting). For each Σ-pattern φ, the lift of φ is a `#Pattern` pattern inductively defined as follows

$$\hat{\varphi} = \begin{cases} \texttt{\#variable}(e_{Var}(x), \hat{s}) & \text{if } \varphi \text{ is a variable } x \in Var_s \subseteq \textsc{Patterns}_s \\ \texttt{\#application}(\hat{\sigma}, (\hat{\psi_1}, \ldots, \hat{\psi_n})) & \text{if } \varphi \text{ is } \sigma(\psi_1, \ldots, \psi_n) \in \textsc{Patterns}_s \\ \texttt{\#\and}(\hat{\psi_1}, \hat{\psi_2}, \hat{s}) & \text{if } \varphi \text{ is } \psi_1 \wedge \psi_2 \in \textsc{Patterns}_s \\ \texttt{\#\not}(\hat{\psi}, \hat{s}) & \text{if } \varphi \text{ is } \neg\psi \in \textsc{Patterns}_s \\ \texttt{\#\exists}(e_{Var}(x), \hat{s_1}, \hat{\psi}, \hat{s_2}) & \text{if } \varphi \text{ is } \exists x.\psi \in \textsc{Patterns}_{s_2} \text{ and } x \in Var_{s_1} \end{cases}$$

(Theory-lifting). Since $T = (S, \Sigma, A)$ is a finite theory, let us suppose

$$S = \{s_1, \ldots, s_n\}, \Sigma = \{\sigma_1, \ldots, \sigma_m\}, A = \{\varphi_1, \ldots, \varphi_k\},$$

are three finite sets. The lift of theory T is a `#Theory` pattern

$$\hat{T} = \texttt{\#theory}(\texttt{\#signature}(\hat{S}, \hat{\Sigma}), \hat{A}),$$

where the lifts of S, Σ, and A are respectively defined as

$$\begin{aligned} \hat{S} &= \hat{s_1}, \ldots, \hat{s_n} \quad \text{is a } \texttt{\#Sort\{\}}\mathit{List} \text{ pattern} \\ \hat{\Sigma} &= \hat{\sigma_1}, \ldots, \hat{\sigma_m} \quad \text{is a } \texttt{\#SymbolList} \text{ pattern} \\ \hat{A} &= \hat{\varphi_1}, \ldots, \hat{\varphi_k} \quad \text{is a } \texttt{\#PatternList} \text{ pattern} \end{aligned}$$

In order to prove Theorem 11, we introduce a canonical model of K.

Definition 14 (Canonical model of K)**.** The canonical model of K, denoted as M_K, contains carrier sets (for each sort in S_K) and relations (for each symbols in Σ_K) that are defined in the following.

The carrier set for the sort *#Pred* is a singleton set $M_{\#Pred} = \{\star\}$.

The carrier set for the sort `#Char` is the set of all 62 constructors of the sort `#Char`, denoted as $M_{\texttt{\#Char}}$.

The carrier set for the sort `#String` is the set of all syntactic patterns of the sort `#String`, denoted as $M_{\texttt{\#String}}$.

The carrier set for the sort `#Sort{}` is the set of all syntactic patterns of the sort `#Sort{}`

$$M_{\texttt{\#Sort}\{\}} = \{\texttt{\#sort}(str) \mid str \in M_{\texttt{\#String}}\}.$$

The carrier set for the sort `#Sort{}`$List$ is the set of all finite lists of $M_{\texttt{\#Sort}\{\}}$:

$$M_{\texttt{\#Sort}\{\}List} = (M_{\texttt{\#Sort}\{\}})^*.$$

The carrier set for the sort `#Symbol` is the set of all syntactic patterns of the sort `#Symbol`

$$M_{\texttt{\#Symbol}} = \{\texttt{\#symbol}(str, l, s) \mid str \in M_{\texttt{\#String}}, l \in M_{\texttt{\#Sort}\{\}List}, s \in M_{\texttt{\#Sort}\{\}}\}.$$

The carrier set for the sort `#SymbolList` is the set of all finite lists over $M_{\texttt{\#Symbol}}$:

$$M_{\texttt{\#SymbolList}} = (M_{\texttt{\#Symbol}})^*.$$

The carrier set for the sort `#Pattern` is the set of all syntactic patterns of the sort `#Pattern`, denoted as $M_{\texttt{\#Pattern}}$.

The carrier set for the sort `#PatternList` is the set of all finite lists over $M_{\texttt{\#Pattern}}$:

$$M_{\texttt{\#PatternList}} = (M_{\texttt{\#Pattern}})^*.$$

The carrier set for the sort `#Signature` is a product set

$$M_{\texttt{\#Signature}} = M_{\texttt{\#Sort}\{\}List} \times M_{\texttt{\#SymbolList}}.$$

The carrier set for the sort `#Theory` is a product set

$$M_{\texttt{\#Theory}} = M_{\texttt{\#Signature}} \times M_{\texttt{\#Pattern}}.$$

The interpretations of most symbols (except `#provable`) in K are so straightforward that they are trivial. For example, `#consSortList` is interpreted as the cons function on $M_{\texttt{\#Sort}\{\}}List$, and `#deletePatternList` is interpreted as a function on $M_{\texttt{\#Pattern}} \times M_{\texttt{\#PatternList}}$ that deletes the first argument from the second, etc.

Basically I want to say that M_K is almost (except $Kprovable$) the initial algebra of K.

The only nontrivial interpretation is the one for `#provable`, as we would like to interpret it *in terms of* matching logic reasoning. The interpretation of `#provable` in the canonical model is a predicate on $M_{\texttt{\#Theory}}$ and $M_{\texttt{\#Pattern}}$. Intuitively, $\texttt{\#provable}_M(T, \varphi)$ holds if both T and φ are well-formed and φ is deducible in the finite matching logic theory T. This intuition is captured by the next formal definition.

For any $T = (\Sigma, A) \in M_{\texttt{\#Theory}}$ and $\varphi \in M_{\texttt{\#Pattern}}$, we define

$$\texttt{\#provable}_M(T, \varphi) = \begin{cases} \emptyset & \text{if } \texttt{\#wellFormed}_M(\Sigma, \varphi) = \emptyset \\ \{\star\} & \text{if } [\![T]\!] \vdash [\![\varphi]\!] \\ \emptyset & \text{if } [\![T]\!] \nvdash [\![\varphi]\!] \end{cases}$$

where the *semantics bracket* $[\![\_]\!]$ is defined (only on well-formed T and φ) as follows:

$[\![T]\!]$ is the matching logic theory $(\Sigma, [\![A]\!])$

$[\![A]\!] = \{[\![\psi]\!] \mid \psi \in A\}$

$$[\![\varphi]\!] = \begin{cases} x:s & \text{if } \varphi \text{ is } \texttt{\#variable}(x,s) \\ \sigma([\![\varphi_1]\!],\ldots,[\![\varphi_n]\!]) & \text{if } \varphi \text{ is } \texttt{\#application}(\sigma,(\varphi_1,\ldots,\varphi_n)) \\ [\![\varphi_1]\!] \wedge [\![\varphi_2]\!] & \text{if } \varphi \text{ is } \texttt{\#\and}(\varphi_1,\varphi_2,s), \\ \neg[\![\varphi_1]\!] & \text{if } \varphi \text{ is } \texttt{\#\not}(\varphi_1,s) \\ \exists x.[\![\varphi_1]\!] & \text{if } \varphi \text{ is } \texttt{\#\exists}(x,s_1,\varphi_1,s_2) \end{cases}$$

It is tedious but straightforward to verify that $\texttt{\#provable}_M$ satisfies all axioms about `#provable` that we introduced in Section **??**. For example, the Rule (K1)

$$\texttt{\#provable}(T, \overline{\varphi \leftrightarrow_s (\psi \leftrightarrow_s \varphi)})$$

holds in K because

$$[\![T]\!] \vdash [\![\varphi]\!] \to ([\![\psi]\!] \to [\![\varphi]\!])$$

The Rule (Modus Ponens)

$$\texttt{\#provable}(T,\varphi) \wedge \texttt{\#provable}(T, \overline{\varphi \leftrightarrow_s \psi}) \to \texttt{\#provable}(T,\psi)$$

holds in K because

$$\text{if } [\![T]\!] \vdash [\![\varphi]\!] \text{ and } [\![T]\!] \vdash [\![\varphi]\!] \to [\![\psi]\!], \text{ then } [\![T]\!] \vdash [\![\psi]\!].$$

Readers are welcomed to verify the remaining axioms in K also hold in M_K. We omit them here.

Now we are in a good shape to prove Theorem 11.

Proof Sketch of Theorem 11.

Step 1 (The "$\Rightarrow$" part).

We prove this by simply mimicking the proof of $T \vdash \varphi$ in K.

Step 2 (The "$\Leftarrow$" part).

Let us fix a finite matching logic theory $T = (S, \Sigma, A)$, a Σ-pattern φ, and an encoding e, and assume that $K \vdash \texttt{\#provable}(\hat{T}, \hat{\varphi})$. Since the matching logic proof system is sound, the interpretation of $\texttt{\#provable}(\hat{T}, \hat{\varphi})$ should hold in the canonical model M_K:

$$\texttt{\#provable}_M(\hat{T}, \hat{\varphi}) = \{\star\}.$$

By definition, this means that

$$[\![\hat{T}]\!] \vdash [\![\hat{\varphi}]\!].$$

Finally notice that by construction of encoding, lifting, and the semantics bracket, there is an isomorphism between T and $[\![\hat{T}]\!]$, so from $[\![\hat{T}]\!] \vdash [\![\hat{\varphi}]\!]$ we have

$$T \vdash \varphi.$$

□

References

[1] D. Bogdănaş and G. Roşu. K-Java: A Complete Semantics of Java. In *POPL'15*, pages 445–456. ACM, January 2015.

[2] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.

[3] A. Ştefănescu, c. Ciobâcă, R. Mereuţă, B. M. Moore, T. F. Şerbănuţă, and G. Roşu. All-path reachability logic. In *RTA-TLCA'14*, volume 8560 of *LNCS*, pages 425–440. Springer, 2014.

[4] A. Ştefănescu, D. Park, S. Yuwen, Y. Li, and G. Roşu. Semantics-based program verifiers for all languages. In *OOPSLA'16*, pages 74–91. ACM, 2016.

[5] C. Ellison and G. Rosu. An executable formal semantics of C with applications. In *POPL*, pages 533–544. ACM, 2012.

[6] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Th. Comp. Sci.*, 103(2):235–271, 1992.

[7] D. Filaretti and S. Maffeis. An executable formal semantics of php. In *ECOOP'14*, LNCS, pages 567–592. Springer, 2014.

[8] D. Guth. A formal semantics of Python 3.3. Master's thesis, University of Illinois at Urbana-Champaign, July 2013. `https://github.com/kframework/python-semantics`.

[9] C. Hathhorn, C. Ellison, and G. Roşu. Defining the undefinedness of C. In *PLDI'15*, pages 336–345. ACM, 2015.

[10] E. Hildenbrandt, M. Saxena, X. Zhu, N. Rodrigues, P. Daian, D. Guth, and G. Rosu. Kevm: A complete semantics of the ethereum virtual machine. Technical Report http://hdl.handle.net/2142/97207, University of Illinois, Aug 2017.

[11] Y. Hirai. Defining the ethereum virtual machine for interactive theorem provers. WSTC17, International Conference on Financial Cryptography and Data Security, 2017.

[12] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Comput. Sci.*, 96(1):73–155, 1992.

[13] D. Park, A. Ştefănescu, and G. Roşu. KJS: A complete formal semantics of JavaScript. In *PLDI'15*, pages 346–356. ACM, 2015.

[14] J. G. Politz, A. Martinez, M. Milano, S. Warren, D. Patterson, J. Li, A. Chitipothu, and S. Krishnamurthi. Python: The full monty. In *OOPSLA'13*, pages 217–232. ACM, 2013.

[15] G. Roşu. CS322 - Programming Language Design: Lecture Notes. Technical Report http://hdl.handle.net/2142/11385, University of Illinois, December 2003.

[16] G. Roşu. Matching logic. *Logical Methods in Computer Science*, to appear, 2017.

[17] G. Roşu, A. Ştefănescu, Ştefan Ciobâcă, and B. M. Moore. One-path reachability logic. In *LICS'13*, pages 358–367. IEEE, 2013.

[18] T. F. Serbanuta and G. Rosu. A truly concurrent semantics for the k framework based on graph transformations. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *ICGT*, volume 7562 of *LNCS*, pages 294–310. Springer, 2012.

[19] B. C. Smith. Reflection and semantics in LISP. In *POPL'84*, pages 23–35. ACM, 1984.

11 Get Material From Here

Matching logic imposes no restrictions on the cardinality of the set of sorts S, of the set of symbols Σ, or of the set of patterns A in a theory (S, Σ, A). In practice, however, we need a finite mechanism to describe such theories in order to mechanically reason about them using a computer. In the vast universe of possibilities, we here propose one particular approach that we found useful, convenient and sufficient in our semantics engineering efforts. Specifically:

- We restrict the infinite sets of sorts to ones which can be described with a finite set of *parametric* sorts, where the parameters range over other sorts. For example, $S = \{Nat, List\{X\}\}$ describes the infinite set of sorts $S = \{Nat, List\{Nat\}, List\{List\{Nat\}\}, ...\}$.

From here on, unless otherwise specified, when we say *set* we mean a *recursively enumerable set*, i.e., a set that can be generated algorithmically (not to be confused with the weaker notion of a countable set, which only means that it has a cardinal no larger than $\aleph_0$—either finite or in bijection with the set of natural numbers).

countable may suffice, we'll need to understand how it is in FOL

In this section, we define a matching logic theory $K = (S_K, \Sigma_K, A_K)$ as *the (reflective) calculus of matching logic*, where S_K, Σ_K, and A_K are *finite* sets of sorts, symbols, and axioms, respectively.

Sorts are declared using the `sort` keyword, symbols are declared using the `symbol` keyword, and axioms are defined using the `axiom` keyword.

```
// Namespaces for sorts, variables, metavariables,
// symbols, and Kore modules.
Sort           = String
VariableId     = String
MetaVariableId = String
Symbol         = String
ModuleId       = String

Variable       = VariableId:Sort
MetaVariable   = MetaVariableId::Sort

Pattern        = Variable | MetaVariable
| \and(Pattern, Pattern)
| \not(Pattern)
| \exists(Variable, Pattern)
| Symbol(PatternList)

Sentence       = import ModuleId
| syntax Sort
```

```
| syntax Sort ::= Symbol(SortList)
| axiom Pattern
Sentences      = Sentence | Sentences Sentences

Module         = module ModuleId
Sentences
endmodule
```

In Kore syntax, the backslash "\" is reserved for matching logic connectives and the sharp "#" is reserved for the meta-level, i.e., the K sorts and symbols. Therefore, the sorts *#Pred*, `#String`, `#Symbol`, `#Sort{}`, and `#Pattern` in the calculus K are denoted as `#Bool`, `#String`, `#Symbol`, `#Sort`, and `#Pattern` in Kore respectively. Symbols in K are denoted in the similar way, too. For example, the constructor symbol `#variable`: `#String` $\times$ `#Sort{}` $\rightarrow$ `#Pattern` is denoted as `#variable` in Kore.

A Kore module definition begins with the keyword `module` followed by the name of the module-being-defined, and ends with the keyword `endmodule`. The body of the definition consists of some *sentences*, whose meaning are introduced in the following.

The keyword `import` takes an argument as the name of the module-being-imported, and looks for that module in previous definitions. If the module is found, the body of that module is copied to the current module. Otherwise, nothing happens. The keyword `syntax` leads a *syntax declaration*, which can be either a *sort declaration* or a *symbol declaration*. Sorts declared by sort declarations are called *object-sorts*, in comparison to the five *meta-sorts*, `#Bool`, `#String`, `#Symbol`, `#Sort`, and `#Pattern`, in K. Symbols whose argument sorts and return sort are all object-sorts (meta-sorts) are called *object-symbols* (*meta-sorts*).

Patterns are written in prefix forms. A pattern is called an *object-pattern* (*meta-pattern*) if all sorts and symbols in it are object (meta) ones. Meta-symbols will be added to the calculus K, while object-sorts and object-symbols will not. They only serve for the purpose to parse an object pattern.

The keyword `axiom` takes a pattern and adds an axiom to the calculus K. If the pattern is a meta-pattern, it adds the pattern itself as an axiom. If the pattern φ is an object-pattern, it adds $[\![\varphi]\!]$ as an axiom to the calculus K.

Recall that we have defined the semantics bracket as

$$[\![\varphi]\!] \equiv (\mathit{deducible}\,(\mathit{lift}[\varphi]) = \mathit{true})\,,$$

where φ is a pattern of the grammar in Figure 1. However, here in Kore we allow φ containing *meta-variables*. As a result, we modify the definition of the semantics bracket as

$$[\![\varphi]\!] \equiv \mathit{mvsc}[\varphi] \rightarrow (\mathit{deducible}\,(\mathit{lift}[\varphi]) = \mathit{true}),$$

where the lifting function $\mathit{lift}[\_]$ and the meta-variable sort constraint $\mathit{mvsc}[\_]$ are defined in Algorithm **??** and 1, respectively. Intuitively, meta-variables in an object-pattern φ are lifted to variables of the sort `#Pattern` with the corresponding sort constraints. For example, the meta-variable $x\texttt{::}s$ is lifted to a variable x`:#Pattern` in K with the constraint that `#getSort(`x`:#Pattern)` $= \mathit{sort}(s)$. The function $\mathit{mvsc}[\_]$ collects all such meta-variable sort constraint in an object-pattern is implemented in Algorithm 1.

Algorithm 1: Meta-Variable Sort Constraint Collection *mvsc*

Input: An object-pattern φ

Output: The meta-variable sort constraint of φ

1 Collect in set W all meta-variables appearing in φ;
2 Let $C = \emptyset$;
3 **foreach** $x::s \in W$ **do**
4 $\quad C = C \cup (sort(s) = \texttt{\#getSort}(x\texttt{:\#Pattern}))$
5 Return $\bigwedge C$;