The Semantics of $\mathbb{K}$

Formal Systems Laboratory
University of Illinois at Urbana-Champaign

March 12, 2018

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 [23]. 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 [4] convenient for teaching programming languages [22], until recently $\mathbb{K}$'s semantics was explained either by translation to rewriting logic [17] or by translation to some form of graph rewriting [25]. 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 [7, 13], Java [3], JavaScript [19], Python [12, 21], PHP [9], EVM [15, 14].

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 $A \vdash \varphi$.

... ...

2 Matching Logic

In this section we first recall basic matching logic syntax and semantics notions from [23] 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".

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

Bring some motivational arguments here why this is all important. Why do we want to define the semantics of $\mathbb{K}$ by going to the meta-level. Why people who just want to implement $\mathbb{K}$ tools should be interested in this document.

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 $\Sigma_{s_1 \ldots s_n, s}$, where $s_1, \ldots, s_n, s \in S$. The formulae of matching logic are called *patterns*, although we may also call them *formulae*. The set PATTERN_s of patterns of sort $s \in S$ is generated by the following grammar:

$$
\begin{array}{lll}
\varphi_s ::= x : s \quad \text{where } x \in \mathit{Name} \text{ and } s \in S & & \text{// variable} \\
\quad \mid \sigma(\varphi_{s_1}, ..., \varphi_{s_n}) \quad \text{where } \sigma \in \Sigma_{s_1 \ldots s_n, s} \text{ and } \varphi_{s_1}, ..., \varphi_{s_n} \text{ of appropriate sorts} & & \text{// structure} \\
\quad \mid \varphi_s \wedge_s \varphi_s & & \text{// intersection} \\
\quad \mid \neg_s \varphi_s & & \text{// complement} \\
\quad \mid \exists_s x : s' . \varphi_s \quad \text{where } x \in N \text{ and } s' \in S & & \text{// binder}
\end{array}
$$

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 context or irrelevant. Similarly, we omit the subscript s of $\wedge_s$, $\neg_s$, and respectively $\exists_s$ when understood from context or irrelevant. Given a symbol $\sigma \in \Sigma_{s_1 \ldots s_n, s}$, $s_1 \ldots s_n$ are called the *argument sorts* of σ, s is called the *return sort* of σ, and n is called the *arity* of σ. By abuse of language, we take the freedom to identify symbols $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ with corresponding patterns $\sigma(x_1 : s_1, ..., x_n : s_n)$, where x_1, ..., x_n are all distinct names. 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 $\{\text{PATTERN}_s\}_{s \in S}$. 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, as seen shortly. For notational convenience, we take the liberty to use mix-fix syntax for operators in Σ and parentheses for grouping. Also, recall that we may omit the sort of a variable when understood. For example, if $\mathit{Nat} \in S$ and $\_ + \_, \_ * \_ \in \Sigma_{\mathit{Nat} \times \mathit{Nat}, \mathit{Nat}}$ then we may write "$(x + y) * z$" instead of "$\_ * \_(\_ + \_(x : \mathit{Nat}, y : \mathit{Nat}), z : \mathit{Nat})$". 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 $\mathit{Mod}(S, \Sigma)$, or $\mathit{Mod}(\Sigma)$ when S is understood, denote the (category of) models of a signature (S, Σ). Given a model M and a map $\rho : \mathit{Var} \to M$, 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 $x \in \mathit{Var}_s$
- $\overline{\rho}(\sigma(\varphi_1, \ldots, \varphi_n)) = \sigma_M(\overline{\rho}(\varphi_1), \ldots \overline{\rho}(\varphi_n))$ 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
- $\overline{\rho}(\exists x.\varphi) = \bigcup\{\overline{\rho'}(\varphi) \mid \rho' : \mathit{Var} \to M,\ \rho' \restriction_{\mathit{Var} \setminus \{x\}} = \rho \restriction_{\mathit{Var} \setminus \{x\}}\} = \bigcup_{a \in M} \overline{\rho[a/x]}(\varphi)$

where " $\setminus$" is set difference, "$\rho\restriction_V$" is ρ restricted to $V \subseteq \mathit{Var}$, 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 $\rho : \mathit{Var} \to M$, 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. A symbol $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ is called a *predicate* if its corresponding pattern $\sigma(x_1 : s_1, ..., x_n : s_n)$ is a predicate.

Model M *satisfies* φ_s, written $M \models \varphi_s$, iff $\overline{\rho}(\varphi_s) = M_s$ for all $\rho : \mathit{Var} \to M$. 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*, or a *(matching logic) theory*, is a triple (S, Σ, A), or just (Σ, A) when S is understood, with A a set of Σ-patterns. We let T range over specification/theories; $T = (S, \Sigma, A)$ is *finite* whenever S, Σ and A are finite. Given specification $T = (S, \Sigma, A)$ we let $\mathit{Mod}(T)$, or $\mathit{Mod}(S, \Sigma, A)$ or $\mathit{Mod}(\Sigma, A)$, also denoted by $[\![T]\!]$, or $[\![(S, \Sigma, A)]\!]$ or $[\![(\Sigma, A)]\!]$, be its (category of) models $\{M \mid M \in \mathit{Mod}_\Sigma,\ M \models_\Sigma A\}$.

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 $M \in \mathit{Mod}(\Sigma)$ then we let $M\restriction_{\Sigma'} \in \mathit{Mod}(\Sigma')$ 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:

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

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 $\varphi \in \text{PATTERN}_{s_1}$ then $\lceil \varphi \rceil^{s_2}_{s_1}$ is a predicate which holds iff φ is defined: if $\rho : Var \to M$ 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 $\varphi \in \text{PATTERN}_{s_1}$ then $\lfloor \varphi \rfloor^{s_2}_{s_1}$ is a predicate where if $\rho : Var \to M$ 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:

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

Equality is also a predicate. if $\varphi, \varphi' \in \text{PATTERN}_{s_1}$ and $\rho : Var \to M$ 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 $x \in Var_{s_1}$, $\varphi \in \text{PATTERN}_{s_1}$ and $s_1, s_2 \in S$, 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 $\rho : Var \to M$, $\overline{\rho}(x \in^{s_2}_{s_1} \varphi) = \emptyset$ iff $\rho(x) \notin \overline{\rho}(\varphi)$, and $\overline{\rho}(x \in^{s_2}_{s_1} \varphi) = M_{s_2}$ iff $\rho(x) \in \overline{\rho}(\varphi)$. It is convenient to extend the definition of membership to cases when the first component is not a variable, but a pattern: if $\psi, \varphi \in \text{PATTERN}_{s_1}$, and $s_1, s_2 \in S$, then let

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

This allows a uniform interface for pattern constructs, which yields neat implementation of operations such as substitution. As an example, consider $\varphi_1[\varphi_2/x]$ where we substitute x for φ_2 in φ_1. If the membership construct only accept variables as its first component, then the substitution is only defined when x does not occur free in the first component of any membership construct in φ_1, and any implementation of substitution will have to take that into account and perform an "occurs check" for x in φ_1, and this could be expensive.

When writing patterns, the following precedence and associativity rules are useful to reduce the number of necessary parentheses.

Connective	Precedence	Associativity
$\neg$	1	–
$=, \in$	2	–
$\wedge$	3	left
$\vee$	4	left
$\rightarrow$	5	right
$\leftrightarrow$	6	–
$\exists,\forall$	7	–

As a convention, binders ($\forall$ and $\exists$) have a lower precedence than other connectives. This means that the scope of a binder extends as far to the right as possible.

Since s_1 and s_2 can usually be inferred from context, we write $\lceil \_ \rceil$ or $\lfloor \_ \rfloor$ instead of $\lceil \_ \rceil_{s_1}^{s_2}$ or $\lfloor \_ \rfloor_{s_1}^{s_2}$, 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 $\lceil x : s_1 \rceil_{s_1}^{s_2}$ 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 "$\sigma : s_1 \times \cdots \times s_n \rightarrow s$" 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*. The functionality property can be extended from a symbol to a pattern: equation "$\exists y \,.\, \varphi = y$" states that pattern φ is *functional*; it is easy to see that terms are functional patterns. Partial functions and total relations can also be axiomatized; we refer the interested reader to [23].

Constructors can also be axiomatized in matching logic. Constructors play a critical role in programming language semantics, because they can be used to build programs, data, as well as semantic structures to define and reason about languages and programs. The main characterizing properties of constructors are "no junk" (i.e., all elements are built with constructors) and "no confusion" (i.e., all elements are built in a unique way using constructors), which can both be defined axiomatically in matching logic. Specifically, let us fix a sort s and suppose that we want to state that the finite set of symbols $\{c_i \in \Sigma_{s_i^1 \ldots s_i^{m_i}, s} \mid 1 \leq i \leq n\}$ are constructors for s. Then

No junk: We require that A contain, or entail, the following pattern

$$\bigvee_{i=1}^{n} \exists x_i^1 : s_i^1 \ldots \exists x_i^{m_i} : s_i^{m_i} \,.\, c_i(x_i^1, \ldots, x_i^{m_i})$$

This states that any element of sort s is in the image of at least one of the constructors.

No confusion, different constructors: For any $1 \leq i \neq j \leq n$, A contains or entails

$$\neg(c_i(x_i^1, \ldots, x_i^{m_i}) \wedge c_j(x_j^1, \ldots, x_j^{m_j}))$$

This states that no element is in the image of two different constructors.

No confusion, same constructor: For any $1 \leq i \leq n$, A contains or entails

$$c_i(x_i^1, \ldots, x_i^{m_i}) \wedge c_i(y_i^1, \ldots, y_i^{m_i}) \rightarrow c_i(x_i^1 \wedge y_i^1, \ldots, x_i^{m_i} \wedge y_i^{m_i})$$

This states that each element is constructed in a unique way. That follows because for any two variables x and y of same sort, $x \wedge y$ is interpreted as either a singleton set when x and y are interpreted as the same element, or as the empty set otherwise.

Additionally, if each c_i is functional, then we call them **functional constructors**. The usual way to define a set of constructors is to have A include all the patterns above.

An interesting observation is that *unification* and, respectively, *anti-unification* (or *generalization*, can be regarded as conjunction and, respectively, as disjunction of patterns [23].

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, for any given finite or infinite specification (Σ, A), a proof system yields a *provability relation*, written $A \vdash \varphi$, 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 [23], 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)
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 $x, y \in Var$ (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. *[23] 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 9, 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:

$\mathit{List}\{s\}$	for any sort s
$\mathit{Set}\{s\}$	for any sort s
$\mathit{Bag}\{s\}$	for any sort s
$\mathit{Bag}_p\{s\}$	for any sort s which is not of the form $\mathit{Bag}_p\{\_\}$
$\mathit{Map}\{s, s'\}$	for any sorts s, s' // for (partial) maps from keys of sort s to values of sort s'
$\mathit{Map}_p\{s, s'\}$	for any sorts s, s' such that s is not of the form $\mathit{Map}_p\{\_, \_\}$
$\mathit{Context}\{s, s'\}$	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 $\mathit{List}\{\mathit{Nat}\}$, $\mathit{Bag}\{\mathit{List}\{\mathit{Nat}\}\}$, $\mathit{Map}\{\mathit{List}\{\mathit{Int}\}, \mathit{Set}\{\mathit{Int}\}\}$, $\mathit{Context}\{\mathit{Set}\{\mathit{Int}\}, \mathit{Map}\{\mathit{Int}, \mathit{Bool}\}\}$, etc. Note that sort schemas, like all schemas, can have *side conditions*. For example, the schema $\mathit{Bag}_p\{s\}$ (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.

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

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

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

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:

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

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:

$\varphi[\varphi_1/x:s] \wedge (\varphi_1 =_s^{s'} \varphi_2) \rightarrow \varphi[\varphi_2/x:s]$	where s, s' are any sorts, φ any pattern of sort s', and φ_1, φ_2 any patterns of sort s
$\forall x:s.\varphi \rightarrow \varphi[t/x]$	where s is any sort and t is any *syntactic pattern*, or *term*, or sort s, i.e., one containing only variables and symbols
$\varphi_1 \texttt{ + } \varphi_2 = \varphi_1 +_{Nat} \varphi_2$	where φ, φ' are *ground* syntactic patterns of sort *Nat*, that is, patterns built only with symbols `zero` and `succ`
$(\varphi_1 \rightarrow \varphi_2) \rightarrow (\varphi[\varphi_1/x] \rightarrow \varphi[\varphi_2/x])$	where φ is a *positive context in* x, that is, a pattern containing only one occurrence of x with no negation ($\neg$) on the path to x, and where φ_1, φ_2 are any patterns having the same sort as x

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 [23]. 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 [23], 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:

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

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. This is not possible in general, because M and ρ are arbitrary and many of the fixed-points of e_M, including the least fixed-point, may very well be unreachable with patterns. However, from a practical perspective, the importance of those syntactically unreachable fixed-points is questionable. 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 syntactic *least fixed-point* nature of $\mu x.e$ with the following pattern schema, which we call "Knaster-Tarski":

Explain why only implication suffices in the LHS.

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

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

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

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 then?

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.

An alternative way to define greatest fixed-point is using the least fixed-point construct:

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

where the pattern $\neg e[\neg x/x]$ is called the *dual of pattern e w.r.t. the variable x*. This definition is justified by the following theorem which establishes the duality between least and greatest fixed-points.

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

Theorem 2 (Duality of Fixed-Points)**.** *Let $F\colon 2^M \to 2^M$ be a monotone function. By Knaster-Tarski theorem, it has a unique least fixed-point $\mu x.F \subseteq M$ and a unique greatest fixed-point $\nu x.F \subseteq M$, which satisfy the following duality equations:*

$$\nu x.F = M \setminus \left(\mu x.\bar{F}\right)$$
$$\mu x.F = M \setminus \left(\nu x.\bar{F}\right)$$

where $\bar{F}\colon 2^M \to 2^M$ is also a monotone function defined as $\bar{F}(A) = M \setminus (F(M \setminus A))$.

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 [8], 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:

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

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 $\mathsf{ite} \in \Sigma_{\mathit{BExp} \times \mathit{Stmt} \times \mathit{Stmt}, \mathit{Stmt}}$, 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:

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

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 [8] 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:

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

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,

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

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,

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

Although $\gamma\Box_1\Box_2\ldots\Box_n.\,T$ correspond to no patterns, we take a freedom to call them *multi-hole contexts* and let meta-variables C range over them, i.e., we take the freedom to write $C[T_1, T_2, \ldots, T_n]$. 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:

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

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

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

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:

$$
\begin{array}{rl}
\mathsf{cfg}(\mathsf{k}(\mathsf{ite}(1 < x, S_1, S_2);\ S_3),\ \mathsf{env}(x \mapsto l, R_{env}),\ \mathsf{mem}(l \mapsto a, R_{mem})) & = \\
\mathsf{cfg}(\mathsf{k}(\mathsf{ite}(1 < x, S_1, S_2);\ S_3),\ \mathsf{env}(x \mapsto l, R_{env}),\ \mathsf{mem}((\gamma\, \square_3\, .\, \square_3)[l \mapsto a], R_{mem})) & = \\
\mathsf{cfg}(\mathsf{k}(\mathsf{ite}(1 < x, S_1, S_2);\ S_3),\ \mathsf{env}(x \mapsto l, R_{env}),\ \mathsf{mem}((\gamma\, \square_3\, .\, (\square_3, R_{mem}))[l \mapsto a])) & = \\
\mathsf{cfg}(\mathsf{k}(\mathsf{ite}(1 < x, S_1, S_2);\ S_3),\ \mathsf{env}(x \mapsto l, R_{env}),\ (\gamma\, \square_3\, .\, \mathsf{mem}(\square_3, R_{mem}))[l \mapsto a]) & = \\
(\gamma\, \square_3\, .\, \mathsf{cfg}(\mathsf{k}(\mathsf{ite}(1 < x, S_1, S_2);\ S_3),\ \mathsf{env}(x \mapsto l, R_{env}),\ \mathsf{mem}(\square_3, R_{mem})))[l \mapsto a] & = \\
\ldots & = \\
(\gamma\, \square_3\, .\, (\gamma\, \square_2\, .\, \mathsf{cfg}(\mathsf{k}(\mathsf{ite}(1 < x, S_1, S_2);\ S_3),\ \mathsf{env}(\square_2, R_{env}),\ \mathsf{mem}(\square_3, R_{mem})))[x \mapsto l])[l \mapsto a] & = \\
(\gamma\, \square_2\, \square_3 .\, \mathsf{cfg}(\mathsf{k}(\mathsf{ite}(1 < x, S_1, S_2);\ S_3),\ \mathsf{env}(\square_2, R_{env}),\ \mathsf{mem}(\square_3, R_{mem})))[x \mapsto l, l \mapsto a] & = \\
\ldots & = \\
(\gamma\, \square_2\, \square_3 .\, (\gamma\, \square_1\, .\, \mathsf{cfg}(\mathsf{k}(\mathsf{ite}(1 < \square_1, S_1, S_2);\ S_3),\ \mathsf{env}(\square_2, R_{env}),\ \mathsf{mem}(\square_3, R_{mem})))[x])[x \mapsto l, l \mapsto a] & = \\
(\gamma\, \square_1\, \square_2\, \square_3\, .\, \mathsf{cfg}(\mathsf{k}(\mathsf{ite}(1 < \square_1, S_1, S_2);\ S_3),\ \mathsf{env}(\square_2, R_{env}),\ \mathsf{mem}(\square_3, R_{mem})))[x, x \mapsto l, l \mapsto a] &
\end{array}
$$

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 [24, 5], 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 $\_ \to_{\mathcal{S}} \_ \subseteq M \times M$ on any given model M, called a *transition system*. The transition system $(M, \to_{\mathcal{S}})$ 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:

$$C[*\varphi_1, \ldots, *\varphi] \rightarrow \circ C[\varphi_1, \ldots, \varphi_n] \qquad \text{where at least one of } * \text{ is } \circ$$

Together with structural framing [23], 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 subpatterns, 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 [6] 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.

4.5 Fake Order-Sorted Algebras

There are various motivations to study order-sorted algebras. Goguen in [10] used order-sorted algebras to treat errors in abstract data types. Many algebraic specifications use subsorts as a mechanism to model subtypes. In $\mathbb{K}$, an order-sorted frontend syntax is used to allow more intuitive and compact $\mathbb{K}$ definitions. Therefore, it might appear that in order to give $\mathbb{K}$ a formal semantics, one has to adopt some kinds of order-sorted algebras as its mathematical foundation.

Decades of years of research proved that order-sorted algebras are intuitive, convenient, and easy to use in many practical cases. However, their power does come for a cost (and sometimes a big cost). Algorithms and decision procedures for order-sorted algebras have been proposed and implemented in languages like Maude [4], but in general order-sorted algebras are hard from a theoretical point of view and reasoning in an order-sorted setting is even harder. On one hand, we allow (and welcome) $\mathbb{K}$ backends to make use of order-sorted decision procedures and algorithms, but at the same time we want to have a crystal clear and easier mathematical foundation for $\mathbb{K}$, and thus we restrict ourselves in matching logic and many-sorted algebras. In other words, *order-sorted syntax is just a frontend feature in* $\mathbb{K}$*, not a backend feature*. In this section, we will introduce techniques[2] to *decorate* a matching logic theory with some additional structures such that it "fakes" and "behaves like" an order-sorted theory. The results in this section can be used directly in defining the underlying matching logic theory of a $\mathbb{K}$ definition.

4.5.1 Fake Subsort Relation

A $\mathbb{K}$ definition defines a sort set S and a partial-order relation $\preceq \subseteq S \times S$ called the *subsort relation*. If $s \preceq s'$, we say s is a *subsort* of s' and s' is a *supersort* of s. The subsort relation is a congruence relation w.r.t. sort parametricity. For example,

$$\begin{array}{rl} \mathit{List}\{s\} \text{ is a subsort of } \mathit{List}\{s'\} & \text{if } s \text{ is a subsort of } s' \\ \mathit{Map}\{s_1, s_2\} \text{ is a subsort of } \mathit{Map}\{s_1', s_2'\} & \text{if } s_1 \text{ is a subsort of } s_1' \text{ and } s_2 \text{ is a subsort of } s_2' \end{array}$$

[2]The techniques are not new. There are many research about the reduction from order-sorted algebras to many-sorted algebras, for example in [11].

In other words, an instance of a parametric sort is the subsort of the other instance if its sort parameters are subsorts of the other's element-wisely. The intuition of $s \preceq s'$ is that any element in sort s is included in sort s'. We fake this intuition of inclusion by defining an injective function called the *injection function*:

$$\begin{aligned} inj\{s, s'\}\colon s \to s' \quad & \text{// a parametric function} \\ & \text{// defined for any } s \preceq s' \\ inj\{s, s\}(x) = x \quad & \text{// (identity) axiom} \\ x = y \to inj\{s, s'\}(x) = inj\{s, s'\} \quad & \text{// (injectivity) axiom} \\ inj\{s, s''\}(x) = inj\{s', s''\}(inj\{s, s'\}(x)) \quad & \text{// (transitivity) axiom} \end{aligned}$$

The dual of injection functions are *retracts*. They are surjective partial functions that are the inverse of injection functions, which can be defined as follows:

$$\begin{aligned} retract\{s', s\}\colon s \rightharpoonup s' \quad & \text{// a parametric partial function} \\ & \text{// defined for any } s \preceq s' \\ retract\{s', s\}(x) = x \quad & \text{// for any } x \text{ is a variable of sort } s' \end{aligned}$$

4.5.2 Fake Overloading

Overloading, or often called *polymorphism* in the context of programming languages, refers to the phenomenon that we use one name for multiple operations. For example,

(a). Use _ :: _ as the cons operation for lists of all types;

(b). Use _ + _ as both the addition of naturals and rationals;

(c). Use _ + _ as both the addition of naturals and concatenation of strings;

In literature, especially literature on order-sorted algebras, researchers have used various terminologies for different types of overloading and polymorphism. In the above example, (a) is often called *parametric overloading*; (b) is called *subsort overloading* if naturals is seen as a subsort of rationals; (c) is called *ad hoc overloading* or *strong overloading*. In general, overloading and polymorphism allow us to write more compact and neat expressions and help to make formal languages look "more human", but those benefits do not come for free.

Xiaohong: We need some concrete discussion about why OS makes things harder. Not even sure it is the right claim because OS people will argue the OS makes things simpler instead of harder.

Our proposal is to make implicit overloading explicit, or in other words, we think that *overloading is a frontend feature, not a backend feature*. We have seen an example in Section 3 where we show how parametric overloading is dealt with by explicit writing the sort parameters. In there, one can think of that there exists a smart frontend parser that allows users to write the following compact expressions in frontends

$$\begin{aligned} 1 :: 2 :: nil \quad & \text{// a list of two naturals } 1, 2 \\ \text{“1”} :: \text{“2”} :: nil \quad & \text{// a list of two strings “1”, “2”} \end{aligned}$$

while it parses those expressions as the follows for backends

$$\begin{aligned} 1 ::\{Nat\}\ 2 ::\{Nat\}\ nil\{Nat\} \quad & \text{// a list of two naturals } 1, 2 \\ \text{“1”} ::\{String\}\ \text{“2”} ::\{String\}\ nil\{String\} \quad & \text{// a list of two strings “1”, “2”} \end{aligned}$$

where we define the following parametric sorts and symbols for lists, and the *nil* and *cons* operations:

$$\begin{aligned} List\{s\} &\quad \text{for any sort } s \in \{Nat, String\} \\ nil\{s\} \in \Sigma_{*,List\{s\}} &\quad \text{for any sort } s \in \{Nat, String\} \\ cons\{s\} \in \Sigma_{s \times List\{s\},List\{s\}} &\quad \text{for any sort } s \in \{Nat, String\} \end{aligned}$$

In this section, we show that subsort overloading can be handled in the similar way as we handle parametric overloading. For simplicity, we assume that every symbol σ that is overloaded in frontend syntax has their multiple instances explicitly defined using sort parameters, and the sort parameters are its argument sorts plus its return sort. For example, we want to overload $\_ + \_$ for both additions of naturals and rationals. Then we need to define all four instances for backend use:

Xiaohong: Alternatively, we can define just the first and the last instances. But then the parser needs to explicitly add injection functions. It is hard to argue which of the following is better to parse expression $1 + 2.5$:
(A) $1+\{Nat, Rat, Rat\}2.5$, with the homomorphism axiom saying that $x+\{Nat, Rat, Rat\}y = inj\{Nat, Rat\}(x)+\{Rat, Rat, Rat\}y$ for any x of sort Nat and y of the sort Rat
(B) $inj\{Nat, Rat\}(1)+\{Rat, Rat, Rat\}2.5$
Notice that (A) needs all four instances to be defined (as shown below), while (B) only needs two (the first one and the last one). I personally am in favor of (A).

$$\begin{aligned} \_+\{Nat, Nat, Nat\}\_ &: Nat \times Nat \to Nat \quad // \text{ the addition for two naturals} \\ \_+\{Nat, Rat, Rat\}\_ &: Nat \times Rat \to Rat \quad // \text{ the addition for two naturals} \\ \_+\{Rat, Nat, Rat\}\_ &: Rat \times Nat \to Rat \quad // \text{ the addition for two naturals} \\ \_+\{Rat, Rat, Rat\}\_ &: Rat \times Rat \to Rat \quad // \text{ the addition for two naturals} \end{aligned}$$

or more compactly defined as follows

$$\begin{aligned} \_+\{s_1, s_2, s_3\}\_ &: s_1 \times s_2 \to s_3 \quad \text{where } s_1, s_2, s_3 \in \{Nat, Rat\} \\ &\qquad\qquad \text{and if one of } s_1 \text{ and } s_2 \text{ is } Rat, \text{ then } s_3 \text{ is } Rat \end{aligned}$$

In general, it is a frontend issue to decide how to parse an expression with overloading (e.g. how to calculate the least sort of an expression in an order-sorted signature), and there are a large volume of research about that. In particular, $\mathbb{K}$ has a frontend syntax that is order-sorted, and thus it also gets its own frontend parser, which might be different from other languages with an order-sorted syntax, say Maude. In this section, we do not constrain ourselves in any particular frontend syntax. Instead, we will provide *a general theory of overloading* that supports any order-sorted frontend syntax. In the following, we assume that all instances of an overloaded symbol are defined properly and are ready for frontend parsers to use. The main technical result we will prove is that under some assumptions, different parses of a frontend expression with overloading are equivalent modulo injection functions.

The following homomorphism axiom should hold for all symbols σ that is overloaded in frontend:

$$inj\{s', s''\}(\sigma\{s'_1, \ldots, s'_n, s'\}(x_1 : s'_1, \ldots, x_n : s'_n)) = \sigma\{s''_1, \ldots, s''_n, s''\}(inj\{s'_1, s''_1\}(x_1 : s'_1), \ldots, inj\{s'_n, s''_n\}(x_n : s'_n))$$

where $s'_i \preceq s''_i$ for any $i = 1, \ldots, n$ and $s' \preceq s''$.

The following needs revised.

The above property is especially useful. It allows us to do *sort casting* however we want to. Together with the strictness axioms for injection functions which we introduce later, it provides us full reasoning power of dealing with order-sorted signatures in a multi-sorted setting. In other words, it tells us how to generate proof objects about sort casting and etc. How to design algorithms and

data structures that make doing sort casting and sort inference (introduced below) more efficient is of course another important question.

Notice that any term t in an order-sorted signature is also a well-formed pattern in the corresponding matching logic signature except that the sort parameters of symbols (that correspond to the operators in the order-sorted signature) are not yet decided. *Sort inference in order-sorted signature* is the process of finding a term t its least sort s in an order-sorted signature. We overload the terminology *sort inference* here to mean the process of finding sort parameters for symbols (that correspond to the order-sorted operators) in a pattern t (that correspond to the term t in the order-sorted signature) so that t is well-formed. Suppose $f(t_1, \ldots, t_n)$ is a term in an order-sorted signature. We want to find sort parameters $s_1, \ldots, s_n$ such that $f\{s_1, \ldots, s_n\}(t_1, \ldots, t_n)$ is a well-formed pattern in matching logic. In fact, we only need to pick s_i to be the least sort of t_i in the order-sorted signature obtained by the sort inference in order-sorted signature.

The same sort inference procedure works not only for symbols but also for logic connectives. Suppose $l \Rightarrow r$ is a rewrite rule where the least sorts of l and r are s_l and s_r respectively. Notice that in front-ends, we may not necessarily ask that s_l and s_r are the same sort but just ask they belong to the same connected component. We denote s as the lub of the least sorts of l and r. Then $l \Rightarrow r$ is the syntactic sugar of the following axiom

$$inj\{s_l, s\}(l) \to_s \circ\{s\} inj\{s_r, s\}(r)$$

Injection function are strict. For any $s_1 \preceq s_2$ and the injection function $inj\{s_1, s_2\}$, the following property holds

$$inj\{s_1, s_2\}(C[T]) = (\gamma\Box.inj\{s_1, s_2\}(C[\Box]))[T]$$

It is often convenient to assume that $(S, \preceq)$ is not just a poset but a lattice. There are standard techniques that extend the poset $(S, \preceq)$ to a lattice by introducing new sorts to S if needed. It is orthogonal to what we have discussed.

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 3. 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 4. 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 Towards a Formal Semantics of $\mathbb{K}$: The Need For a Meta-Theory

$\mathbb{K}$ is a best effort realization of matching logic. It was created and developed as a semantic framework for the domain of programming languages, but it is not limited to this domain. It was, however, tuned for defining and reasoning with theories like those in Section 4, i.e., binders, fixed-points, contexts, rewriting, and reachability, which are particularly useful for programming language semantics. In the context of complex languages, we often need many or all of the above-mentioned theories in addition to more user-defined theories, that is, proof tasks have the form:

$$\underbrace{T_{\text{binders}} \cup T_{\text{fixed-points}} \cup T_{\text{contexts}} \cup T_{\text{rewriting}} \cup \cdots \cup T_{\text{user-defined}} \vdash \cdots}_{\mathbb{K} \text{ is a best effort realization of matching logic reasoning}}$$

Notice that, as discussed in Section 4, all these theories are potentially infinite. To make it possible to even write down such theories or proof tasks, $\mathbb{K}$ has adopted finite mechanisms to define infinite theories like those in Section 3 based on sort, symbol, and pattern *schemas*.

This leads to the question: *what is the formal semantics of those schema mechanisms, and thus of* $\mathbb{K}$? Without a precise answer to this question, the problem of a correct implementation of $\mathbb{K}$ cannot even be formulated. Even without the correctness of the $\mathbb{K}$ implementation explicitly stated as a goal, developers of $\mathbb{K}$ would have to guess what those schema mechanisms mean and thus yield potential inconsistencies, disagreements and ultimately slowdowns in the implementation process.

It is worth noting that any formalism expressive enough to capture the intended meaning of schemas would suffice for this purpose. For example, one could choose a formal system like Coq [2], Isabelle [20], Agda [18], Vampire [16], Maude [4], etc., and encode all matching logic elements in it: sorts and sort schemas, symbols and symbol schemas, patterns and pattrn schemas, and even the entire sound and complete deduction system in Section 2.4. Then one could use such a target system not only to yield a formal semantics to finite or infinite matching logic theories and thus

to $\mathbb{K}$, but also to define and reason about any matching logic theory. Such encodings, also called embeddings, of one formalism into another are quite common. In fact, previous implementations of $\mathbb{K}$ used two such encodings, one to Maude and another to Coq (called $\mathbb{K}$ backends).

One important lesson we learned from our previous embedding efforts of $\mathbb{K}$ into other systems was that such encodings can be quite complex and slow, hard to understand and thus to maintain and debug, and ultimately depend upon the target system which itself is a research tool under continuous change/improvement. Therefore, based on our experience, such encoding can hardly be used as a means to describing the semantics of $\mathbb{K}$. Interested users or developers of $\mathbb{K}$ would not only have to learn an additional language and system, the target system, but would also have to decipher complex encodings and translate their target language meaning back to $\mathbb{K}$. Consequently, we take a different approach here, which is also relatively common: *we define the formal semantics of matching logic in (a fragment of) matching logic itself*. Namely, the *finite* fragment of matching logic, which is much simpler and is well-understood.

Specifically, in Section 7 we define a particular but important finite matching logic theory K, which we call *the meta-theory of matching logic*, with the property that it can represent and reason about any (recursively enumerable) elements of matching logic, finite or infinite. In particular, K enjoys the following *reflection* property, which we split in two properties for clarity and terminology:

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

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

where

- T is any *recursively enumerable* matching logic theory, including all (finite or infinite) theories in Section 4;
- $[\![T]\!]$, called the *meta-representation of* T, consists of definitions of finitely many matching logic symbols and finitely many matching logic patterns as axioms;
- $K \cup [\![T]\!]$, called the *meta-theory instantiated by* T, is obtained by adding the new symbols and axioms in $[\![T]\!]$ to the meta-theory K;
- $[\![\varphi]\!]$ is the *meta-representation of* φ, and it is a matching logic pattern in $K \cup [\![T]\!]$, i.e., the meta-theory instantiated by T;
- `#provable` is a predicate symbol defined in the meta-theory K that axiomatizes the provability relation of matching logic (see Section 7.7);

With the meta-theory K, any schema mechanisms, conventions or notations that are found useful to define theories T can be given a formal semantics by defining a corresponding meta-representation $[\![T]\!]$, which is a finite artifact. Although any recursively enumerable domain can be algorithmically represented as a finite equational theory [1], and thus as a finite matching logic theory, for practical reasons we consider new types of recursively enumerable theories T and their encodings $[\![T]\!]$ on a by-need basis. In other words, the meta-theory K gives us a one-stop solution to designing and using new mechanisms and notations to define potentially infinite theories.

From a practical perspective, the meta-theory K can be regarded as bootstrapping and thus as a way to incentivize an accelerated development of implementations of matching logic, such as $\mathbb{K}$. Moreover, since the meta-theory K is ultimately a finite matching logic theory, defined by simply enumerating all its sorts, symbols and patterns (Section 7), it is relatively easy to implement and translate to other formalisms, so it can serve as a starting point for implementing tools and backends for $\mathbb{K}$. Even if tools/backends for $\mathbb{K}$ prefer a different implementation approach, the meta-theory K still serves as a uniform foundation/language which they can use to justify their correctness or communicate artifacts, e.g., generated theories or proofs.

It is worth noting that the meta-theory K makes matching logic a *reflective logic*; according to [26], reflection is "an entity's 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." While reflection is an interesting property to have for all the reasons above and more, note that almost all non-trivial logics enjoy it; for example, all logics including or capable of defining equational logic [1]. The challenge is to devise a *useful meta-theory* that gives tool developers incentives to incorporate it in their tool chain or at least to use it for communicating their tool artifacts. This was the main driving force behind the design of our K meta-theory presented in Section 7.

We should also talk about the negatives of using a meta-theory as a reference semantics. E.g., there is apparent additional level of indirection, which goes in both directions; but this is unavoidable, otherwise we get inconsistencies: substitution-like operation needs FV, and $\{x\} = FV(x * 0) = FV(0) = \emptyset$; $[\![\ldots]\!]$ cannot be a symbol.

7 Meta-Theory of Matching Logic

In this section, we present the meta-theory K as a finite matching logic theory[3] $K = (S_K, \Sigma_K, A_K)$ with S_K a finite set of sorts, Σ_S a finite set of symbols, and A_K a finite set of axioms. A summary of the meta-theory K is provided in Section 7.8. In Section 11 we propose a *canonical model* for K, needed to prove a faithfulness theorem. Here we only focus on the syntactic definition of K.

7.1 Warming-Up

The meta-theory K is a complex theory. 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 will be defined when they are firstly used. The others are general and are not specific to K. In this subsection we give an overview of our notations, conventions and techniques used to define K.

7.1.1 Variable Names

Recall that the matching logic grammar that we introduced in Figure 1 assumes an arbitrary but fixed infinite set *Name* of variable names. Generally speaking, it does not matter what 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 make a big difference. We assume the following letters and their variants (primed, subscripted, etc.) are in the set *Name* of variable names:

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

Starting with 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 like in Section 3. This is intended.

[3] Also known as a deep embedding of matching logic in itself.

7.1.2 Three Ways of Defining Predicate Symbols

Recall from Section 2.2 that predicate symbols in matching logic are symbols whose interpretations in models are either the empty set or the total set. We can also capture the main property of a predicate symbol syntactically: symbol $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ is a predicate symbol in a theory T iff

$$T \vdash \forall x_1 \ldots x_n.((\sigma(x_1, \ldots, x_n) = \top) \vee (\sigma(x_1, \ldots, x_n) = \bot))$$

So far, we identified three practical approaches to define predicate symbols. To illustrate these, we use the predicate symbol *isEmpty*, which checks whether a list is empty or not; assume the sort *List* for lists and ignore the irrelevant sort of the list elements. The important characteristic of predicate symbols in matching logic is that we want to be able to use them in *any* sort context.

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

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

and two axioms

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

Assuming that *nil* and *cons* are constructors, as described in Section 2.3, we can prove that *isEmpty* is indeed a predicate symbol. But it has a specific return sort, *Pred*, so it cannot be directly used in any sort context. To do it, we have to introduce the following abbreviation:

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

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

The second approach is to define

$$\mathit{isEmpty}\colon \mathit{List} \to \mathit{Bool}$$

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

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

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:

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

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 *isEmpty* as a *predicate symbol schema*, i.e., as a set of symbols

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

with two axiom schemas:

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

Now *isEmpty* is a *symbol schema*, so it can be put in any sort contexts by simply instantiating the sort parameter s with corresponding sorts.

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 Schemas do not Necessarily Make a Finite (Meta-)Theory Infinite

As seen in Section 7.1.2, the meta-theory K is planned to contain some predicate symbols defined using symbol and pattern/axiom schemas. The reader may wonder whether that leads to an infinite theory. The answer is no. Adding a finite number of symbol schemas to an otherwise finite theory, be it K or not, does not make it an infinite theory. If a theory has only a finite number of sorts, then a symbol schema also has only a finite number of symbol instances, one for each combination of sorts as instances of its parameters (i.e., sort meta-variables). The same applies for axiom/pattern schemas whose meta-variables only range over sorts, like those needed to define the predicate symbols. (Sort schemas, and axiom/pattern schemas with meta-variables ranging over patterns, on the other hand, make a theory infinite.) Therefore, the meta-theory K stays finite as long as the number of sorts are finite (in particular, no sort schemas) and the symbol and axiom/pattern schemas only use sort meta-variables.

7.1.4 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.

7.1.5 A Roadmap of Meta-Theory

We organize the definition of the meta-theory K into six parts, running from Section 7.2 to Section 7.7, as follows:

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 then gives a summary of all sorts, symbols, and axioms of K. We tried to ensure 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. Unfortunately, that was not entirely possible; informal explanations will be given whenever sorts or symbols are used before they are formally defined.

7.2 Characters and Strings

The sort `#Char` is the sort for *characters*. It has the following $26 + 26 + 10 + 5 = 67$ functional constructors (more can be added if needed or desired):

$$
\begin{array}{lll}
\texttt{`a'}: \to \texttt{\#Char} & \texttt{`b'}: \to \texttt{\#Char} & \texttt{`c'}: \to \texttt{\#Char} \\
\quad \cdots & \quad \cdots & \quad \cdots \\
\texttt{`x'}: \to \texttt{\#Char} & \texttt{`y'}: \to \texttt{\#Char} & \texttt{`z'}: \to \texttt{\#Char} \\
\texttt{`A'}: \to \texttt{\#Char} & \texttt{`B'}: \to \texttt{\#Char} & \texttt{`C'}: \to \texttt{\#Char} \\
\quad \cdots & \quad \cdots & \quad \cdots \\
\texttt{`X'}: \to \texttt{\#Char} & \texttt{`Y'}: \to \texttt{\#Char} & \texttt{`Z'}: \to \texttt{\#Char} \\
\texttt{`0'}: \to \texttt{\#Char} & \cdots & \texttt{`9'}: \to \texttt{\#Char} \\
\texttt{`\#'}: \to \texttt{\#Char} & \texttt{`\textbackslash'}: \to \texttt{\#Char} & \texttt{``'}: \to \texttt{\#Char} \\
\texttt{`-'}: \to \texttt{\#Char} & \texttt{`''}: \to \texttt{\#Char} &
\end{array}
$$

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

$$
\begin{array}{l}
\texttt{\#nilCharList}: \to \texttt{\#CharList} \\
\texttt{\#consCharList}: \texttt{\#Char} \times \texttt{\#CharList} \to \texttt{\#CharList}
\end{array}
$$

Notation 5. 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 double-quotation marks. For example, we simply write "`abc`" instead of

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

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 parameters. This unified view of parametric and non-parametric sorts helps to keep the meta-theory simple. We will adopt the same view in dealing with parametric/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. Non-parametric symbols such as *zero* and *plus* are regarded as parametric symbols *zero*{} and *plus*{} taking and empty list of sort parameters.

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

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

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 $x\colon s$ for logic variables when the sort s is understood from the context. Here we are writing

f	as a shorthand of $f\colon\texttt{\#String}$	which is a variable of sort `#String`
S	as a shorthand of $S\colon\texttt{\#SortList}$	which is a variable of sort `#SortList`
S'	as a shorthand of $S'\colon\texttt{\#SortList}$	which is a variable of sort `#SortList`
s	as a shorthand of $s\colon\texttt{\#Sort}$	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* symbols, are required by the sound and complete proof system of matching logic [23]. For any matching logic theory and two sorts s, s' in the theory, we assume the definedness symbol $\lceil \_ \rceil_s^{s'}$ is defined in the theory, and it has an axiom $\lceil x\colon s \rceil_s^{s'}$. 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 and the linear representations.

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

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

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, via the 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:`#Sort`
s'	as a shorthand of s':`#Sort`
(s, s')	as a shorthand of `#consSortList`$(s,$ `#consSortList`$(s',$ `#nilSortList`$))$
(s)	as a shorthand of `#consSortList`$(s,$ `#nilSortList`$)$

The last two shorthands are defined in Notation 7.

Remark 6. The function `#‘ceil` is used to construct the meta-representations of definedness symbols *in the object theories*. The reader may have noticed that we assumed equality defined in the meta-theory K, which means that we also assumed definedness symbols defined. In fact, as discussed in Section 7.1.3, the meta-theory K has n^2 different definedness symbols:

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

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.

An explanation for the name `#‘ceil` is needed. The first character of the name, `#`, tells us that it belongs to the meta-theory K. The second character, `‘`, tells us that it corresponds to something defined in the object theory (in this case, the definedness symbols in object theories). As a general principle, we use the quote `‘` 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 in Sections 7.2 and 7.3, respectively, the meta-theory K has a sort `#CharList` for cons-lists of characters and a sort `#SortList` for cons-lists of sorts. 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 already defined the sorts `#Char`, `#Sort`, and `#Symbol`. The sorts `#Variable` and `#Pattern` will be defined shortly in Section 7.5. The five types of cons-lists have similar definitions. We only show one of them below, namely how to define cons-lists of sorts.

7.4.1 Finite Lists of Sorts

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

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

The function

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

takes two lists and returns their concatenation, and can be defined as follows (assume s is a `#Sort` variable and S_0, S are `#SortList` variables):

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

For any sort $\#s \in S_K$ of the meta-theory, we define a predicate symbol

$$\texttt{\#inSortList}\{\#s\} \colon \texttt{\#Sort} \times \texttt{\#SortList} \to \#s$$

that takes a sort and a cons-list of sorts and checks list membership. The predicate symbol $\texttt{\#inSortList}\{\#s\}$ yields either $\top_{\#s}$ or $\bot_{\#s}$ so that it can be used as a pattern of sort $\#s$ (see Section 7.1.2 and Section 7.1.3). Its semantics can be defined with the following two axioms for each $\#s \in S_K$ — as discussed in Section 7.1.3, we only have a finite number of instances (here s, s' are `#Sort` variables, and S is a `#SortList` variable):

$$\neg\texttt{\#inSortList}\{\#s\}(s, \texttt{\#nilSortList})$$
$$\texttt{\#inSortList}\{\#s\}(s, \texttt{\#consSortList}(s', S)) = (s = s') \vee \texttt{\#inSortList}\{\#s\}(s, S)$$

The functional

$$\texttt{\#deleteSortList} \colon \texttt{\#Sort} \times \texttt{\#SortList} \to \texttt{\#SortList}$$

takes a sort and a sort list, and returns the sort list in which all the occurrences of the sort are deleted, and the order of the remaining elements does not change. It has two axioms

$$\texttt{\#deleteSortList}(s, \texttt{\#nilSortList}) = \texttt{\#nilSortList}$$
$$\texttt{\#deleteSortList}(s, \texttt{\#consSortList}(s', S))$$
$$= ((s = s') \wedge \texttt{\#deleteSortList}(s, S)) \vee ((s \neq s') \wedge \texttt{\#consSortList}(s', \texttt{\#deleteSortList}(s, S)))$$

Notation 7. To write list expressions more compactly, we use abbreviation(s)

$$(s_1) \equiv \texttt{\#consSortList}(s_1, \texttt{\#nilSortList})$$
$$(s_1, s_2, \ldots, s_n) \equiv \texttt{\#consSortList}(s_1, \texttt{\#consSortList}(s_2, \ldots \texttt{\#consSortList}(s_n, \texttt{\#nilSortList}) \ldots))$$

We may refer to this abbreviation as the *mixfix form representation of lists*.

7.4.2 Finite Lists of Characters, Symbols, Variables, and Patterns

As already mentioned, the cons-lists of characters, symbols, variables, and patterns can be defined the same way we defined the cons-lists of sorts above. For example, for cons-lists of patterns we add sort `#PatternList`, symbols `#nilPatternList`, `#consPatternList`, `#inPatternList`{#s}, etc.

7.5 Matching Logic Patterns

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

$$\texttt{\#variable} \colon \underbrace{\texttt{\#String}}_{\text{variable name}} \times \underbrace{\texttt{\#Sort}}_{\text{variable sort}} \to \texttt{\#Variable}$$

Matching Logic Patterns. The sort `#Pattern` is the sort for matching logic patterns. Recall the grammar of matching logic in Figure 1. There are five production rules in the grammar: Variable, Symbol Application, Conjunction $\wedge_s$, Negation $\neg_s$, and (Existential) Quantification $\exists_s$. Conjunction, Negation, and Quantification are in fact parametric on a sort s. Correspondingly, the sort `#Pattern` has five constructors: the following four functions

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

and an injection function from variables to patterns

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

The following axiom says `#variableAsPattern` is injective:

$$v_1 \neq v_2 \to \texttt{\#variableAsPattern}(v_1) \neq \texttt{\#variableAsPattern}(v_2)$$

Recall that in Section 2.1, we use *Var* to denote the set of all variables, and Pattern to denote the set of all patterns, when a matching logic signature is given. In there, by definition, we have $\mathit{Var} \subseteq$ Pattern because any variable is also a pattern. However, in the meta-theory, we decided to define `#Variable` and `#Pattern` as two different sorts, and use the injective function `#variableAsPattern` to mimic the inclusion relation $\mathit{Var} \subseteq$ Pattern. A related function is

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

It has axiom

$$\texttt{\#variablePattern}(x, s) = \texttt{\#variableAsPattern}(\texttt{\#variable}(x, s))$$

In matching logic, apart from the five productions in the grammar, we also introduce many *derived constructs* as syntactic sugar to help us write compact formulae, such as Disjunction $\vee_s$, Equality $=^{s_2}_{s_1}$, and Membership $\in^{s_2}_{s_1}$. In practice, we found it is often useful to give those derived connectives a correspondence in the meta-theory. Therefore, the sort `#Pattern` also has the following functions (not constructors) in correspondence to those derived connectives

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

We refer readers to Section 2.3 for the reasons why the membership construct "`#\in`" requires sort `#Pattern` instead of sort `#Variable` for its third argument.

Notation 8. As a convention, we are writing

x	as a shorthand of x:`#String`	which is a variable of sort `#String`
y	as a shorthand of y:`#String`	which is a variable of sort `#String`
z	as a shorthand of z:`#String`	which is a variable of sort `#String`
s	as a shorthand of s:`#Sort`	which is a variable of sort `#Sort`
σ	as a shorthand of σ:`#Symbol`	which is a variable of sort `#Symbol`
v	as a shorthand of v:`#Variable`	which is a variable of sort `#Variable`
u	as a shorthand of u:`#Variable`	which is a variable of sort `#Variable`
φ	as a shorthand of φ:`#Pattern`	which is a variable of sort `#Pattern`
ψ	as a shorthand of ψ:`#Pattern`	which is a variable of sort `#Pattern`

Derived connectives have axioms

$$\texttt{\#\or}(s,\varphi,\psi) = \texttt{\#\not}(s,\texttt{\#\and}(s,\texttt{\#\not}(s,\varphi),\texttt{\#\not}(s,\psi)))$$
$$\texttt{\#\implies}(s,\varphi,\psi) = \texttt{\#\or}(s,\texttt{\#\not}(s,\varphi),\psi)$$
$$\texttt{\#\iff}(s,\varphi,\psi) = \texttt{\#\and}(s,\texttt{\#\implies}(s,\varphi,\psi),\texttt{\#\implies}(s,\psi,\varphi))$$
$$\texttt{\#\forall}(s,v,\varphi) = \texttt{\#\not}(s,\texttt{\#\exists}(s,v,\texttt{\#\not}(s,\varphi)))$$
$$\texttt{\#\ceil}(s_1,s_2,\varphi) = \texttt{\#application}(\texttt{\#`ceil}(s_1,s_2),\underbrace{(\varphi)}_{\text{the singleton list that contains only } \varphi})$$
$$\texttt{\#\floor}(s_1,s_2,\varphi) = \texttt{\#\not}(s_2,\texttt{\#\ceil}(s_1,s_2,\texttt{\#\not}(s_1,\varphi)))$$
$$\texttt{\#\equals}(s_1,s_2,\varphi,\psi) = \texttt{\#\floor}(s_1,s_2,\texttt{\#\iff}(s_1,\varphi,\psi))$$
$$\texttt{\#\in}(s_1,s_2,\varphi,\psi) = \texttt{\#\ceil}(s_1,s_2,\texttt{\#\and}(s_1,\varphi,\psi))$$
$$\texttt{\#\top}(s) = \texttt{\#\exists}(s,\texttt{\#variable}(x,s),\texttt{\#variablePattern}(x,s))$$
$$\texttt{\#\bottom}(s) = \texttt{\#\not}(s,\texttt{\#\top}(s))$$

Notation 9. As one may have already noticed, patterns of sort `#Pattern` get huge rather quickly. The following notations are adopted to write `#Pattern` patterns in a more compact way, by putting a bar over their normal mixfix forms

We should define this overbar notation in a recursive manner.

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

Apart from the five constructors and ten derived connectives introduced above, the sort `#Pattern` also gets some common utility functions. Those utility functions are defined in the following subsections.

Free Variable Collection. The function

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

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 is

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

which takes a list of patterns and applies `#getFV` on each of them, and returns the concatenation of the results. Assume L, R are `#PatternList` variables. The next seven axioms define functions `#getFV` and `#getFVFromPatterns`, with the first five of them defining `#getFV` and the last two defining `#getFVFromPatterns`

$$\texttt{\#getFV}(v) = \texttt{\#consVariableList}(v, \texttt{\#nilVariableList})$$
$$\texttt{\#getFV}(\overline{\sigma(L)}) = \texttt{\#getFVFromPatterns}(L)$$
$$\texttt{\#getFV}(\overline{\varphi \wedge_s \psi}) = \texttt{\#appendVariableList}(\texttt{\#getFV}(\varphi), \texttt{\#getFV}(\psi))$$
$$\texttt{\#getFV}(\overline{\neg_s \varphi}) = \texttt{\#getFV}(\varphi)$$
$$\texttt{\#getFV}(\overline{\exists_s v.\varphi}) = \texttt{\#deleteVariableList}(v, \texttt{\#getFV}(\varphi))$$
$$\texttt{\#getFVFromPatterns}(\texttt{\#nilPatternList}) = \texttt{\#nilVariableList}$$
$$\texttt{\#getFVFromPatterns}(\texttt{\#consPatternList}(\varphi, L))$$
$$= \texttt{\#appendVariableList}(\texttt{\#getFV}(\varphi), \texttt{\#getFVFromPatterns}(L))$$

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. It has axiom

$$\texttt{\#occursFree}\{\#s\}(v, \varphi) = \texttt{\#inVariableList}\{\#s\}(v, \texttt{\#getFV}(\varphi))$$

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 a list of patterns. Generally speaking, in matching logic, given a list of patterns $\varphi_1, \ldots, \varphi_n$, a variable name x is said to be a *fresh name* if and only if for any sort s, the variable $x\colon s$ does not occur free in $\varphi_1, \ldots, \varphi_n$. This leads to the following axiom schema

$$\neg(\texttt{\#inVariableList}\{\#s\}(\texttt{\#variable}(\texttt{\#freshName}(L), s), \texttt{\#getFVFromPatterns}(L)))$$

Substitution. The function

$$\texttt{\#substitute}\colon \underbrace{\texttt{\#Pattern}}_{\text{a target pattern } \varphi} \times \underbrace{\texttt{\#Pattern}}_{\text{a ``replace''-pattern } \psi} \times \underbrace{\texttt{\#Variable}}_{\text{a ``find''-variable } v} \to \underbrace{\texttt{\#Pattern}}_{\text{substitute } \psi \text{ for } x \text{ in } \varphi}$$

substitutes ψ for x in φ and returns the substitution result $\varphi[\psi/v]$. A related function is

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

which substitutes ψ for x in a list of patterns. We define the familiar overbar abbreviation

$$\overline{\varphi[\psi/v]} \equiv \texttt{\#substitute}(\varphi, \psi, v)$$
$$\overline{(\varphi_1, \ldots, \varphi_n)[\psi/v]} \equiv \texttt{\#substitutePatterns}((\varphi_1, \ldots, \varphi_n), \psi, v)$$

Recall our naming conventions in Notation 8. Assume L is a `#PatternList` variable. The functions `#substitute` and `#substitutePatterns` have axioms

$$\overline{u[\psi/v]} = ((u = v) \wedge \psi) \vee ((u \neq v) \wedge \texttt{\#variableAsPattern}(u))$$
$$\overline{\sigma(L)[\psi/v]} = \overline{\sigma(L[\psi/v])}$$
$$\overline{(\varphi_1 \wedge_s \varphi_2)[\psi/v]} = \overline{\varphi_1[\psi/v]} \wedge_s \overline{\varphi_2[\psi/v]}$$
$$\overline{(\neg_s \varphi)[\psi/v]} = \overline{\neg_s \varphi[\psi/v]}$$
$$\overline{(\exists_{s'} x\colon s.\varphi)[\psi/v]} \qquad \text{// this complex axiom is needed to prevent free variable capturing}$$
$$= \exists x'.\left(x' = \texttt{\#freshName}(\varphi, \psi, \texttt{\#variableAsPattern}(v)) \wedge \overline{\exists_{s'} x'\colon s.(\varphi[x'\colon s/x\colon s][\psi/v])}\right)$$
$$\overline{\texttt{\#nilPatternList}[\psi/v]} = \texttt{\#nilPatternList}$$
$$\overline{\texttt{\#consPatternList}(\varphi, L)[\psi/v]} = \texttt{\#consPatternList}(\overline{\varphi[\psi/v]}, \overline{L[\psi/v]})$$

Alpha-Renaming and Alpha-Equivalence. In matching logic, alpha-renaming is always assumed. This means that given a matching logic signature, the set of patterns is the one generated by the grammar of matching logic (Figure 1) *modulo alpha-renaming*. In other words, matching logic patterns are equivalence classes with respect to alpha-equivalence. This fact is captured by the next axiom schema

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

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 or declared in the *current theory* and what are not. They are

$$\begin{aligned}&\texttt{\#sortDeclared}\{\#s\}\colon\texttt{\#Sort}\rightarrow\#s\\&\texttt{\#symbolDeclared}\{\#s\}\colon\texttt{\#Symbol}\rightarrow\#s\\&\texttt{\#axiomDeclared}\{\#s\}\colon\texttt{\#Pattern}\rightarrow\#s\end{aligned}$$

A related predicate symbol

$$\texttt{\#sortsDeclared}\{\#s\}\colon\texttt{\#SortList}\rightarrow\#s \tag{1}$$

checks whether a list of sorts are declared:

$$\begin{aligned}&\texttt{\#sortsDeclared}\#s\colon\texttt{\#SortList}\rightarrow\#s\\&\texttt{\#sortsDeclared}\#s(\texttt{\#nilSortList})\\&\texttt{\#sortsDeclared}\#s(\texttt{\#consSortList}(s,S))\\&=\texttt{\#sortDeclared}\{\#s\}(s)\wedge\texttt{\#sortsDeclared}\{\#s\}(S)\end{aligned}$$

These predicate symbols give us what we call an *intensional definition* of matching logic theories.

Extensional and Intensional Definitions. When we define a matching logic theory $T=(S,\Sigma,A)$, we are mainly defining three sets: a set S of sorts, a set Σ of symbols, and a set A of axioms. There are in general two ways to define a set: either extensionally or intensionally. For example, one is to define a set P of all prime numbers. An extensional definition would be

$$P=\{2,3,5,7,11,\dots\}$$

in which one enumerates all prime numbers. An intensional definition would be

$$P=\{n\in\mathbb{N}\mid\text{the only factors }n\text{ has is 1 and }n\text{ itself}\}$$

in which one uses a necessary and sufficient condition to specify what numbers are prime numbers. Extensional definitions are preferred when the sets being defined are small finite sets, while intensional definitions are easier to use when the sets being defined have infinitely many elements and have a simple necessary and sufficient specification condition. In the meta-theory K, we prefer to use intensional definitions to define the sort, symbol, and axiom sets, because they are often infinite sets and we cannot afford enumerating all of their elements.

Definedness Symbols are Defined in All Theories. The definedness symbol $\lceil \_ \rceil_{s_1}^{s_2}$ is always declared if s_1 and s_2 are declared in the current theory:

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

Wellformed Patterns. The predicate symbol schema

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

decides whether a pattern is a *wellformed pattern*. A related predicate symbol

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

decides whether multiple patterns are all wellformed, which has 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 $\texttt{\#wellFormed}\{\#s\}$ and $\texttt{\#getSort}$ are defined by the following axioms

$$\texttt{\#wellFormed}\{\#s\}(\overline{x:s}) = \texttt{\#sortDeclared}\{\#s\}(s)$$

$$\texttt{\#wellFormed}\{\#s\}(\overline{\sigma(L)}) = \texttt{\#symbolDeclared}\{\#s\}(\sigma) \wedge \texttt{\#wellFormedPatterns}\{\#s\}(L)$$
$$\wedge\, \texttt{\#sortDeclared}\{\#s\}\{\texttt{\#getReturnSort}(\sigma)\}$$
$$\wedge\, (\texttt{\#getSortsFromPatterns}(L) = \texttt{\#getArgumentSorts}(\sigma))$$

$$\texttt{\#wellFormed}\{\#s\}(\overline{\varphi \wedge_s \psi}) = \texttt{\#wellFormed}\{\#s\}(\varphi) \wedge \texttt{\#wellFormed}\{\#s\}(\psi)$$
$$\wedge\, (\texttt{\#getSort}(\varphi) = s) \wedge (\texttt{\#getSort}(\psi) = s)$$

$$\texttt{\#wellFormed}\{\#s\}(\overline{\neg_s \varphi}) = \texttt{\#wellFormed}\{\#s\}(\varphi) \wedge (\texttt{\#getSort}(\varphi) = s)$$

$$\texttt{\#wellFormed}\{\#s\}(\overline{\exists_s v.\varphi}) = \texttt{\#wellFormed}\{\#s\}(\texttt{\#variableAsPattern}(v))$$
$$\wedge\, \texttt{\#wellFormed}\{\#s\}(\varphi) \wedge (\texttt{\#getSort}(\varphi) = s)$$

$$\texttt{\#getSort}(\overline{x:s}) = \texttt{\#wellFormed}\{\texttt{\#Sort}\}(\overline{x:s}) \wedge s$$

$$\texttt{\#getSort}(\overline{\sigma(L)}) = \texttt{\#wellFormed}\{\texttt{\#Sort}\}(\overline{\sigma(L)}) \wedge \texttt{\#getReturnSort}(\sigma)$$

$$\texttt{\#getSort}(\overline{\varphi \wedge_s \psi}) = \texttt{\#wellFormed}\{\texttt{\#Sort}\}(\overline{\varphi \wedge_s \psi}) \wedge s$$

$$\texttt{\#getSort}(\overline{\neg_s \varphi}) = \texttt{\#wellFormed}\{\texttt{\#Sort}\}(\overline{\neg_s \varphi}) \wedge s$$

$$\texttt{\#getSort}(\overline{\exists_s v.\varphi}) = \texttt{\#wellFormed}\{\texttt{\#Sort}\}(\overline{\exists_s v.\varphi}) \wedge s$$

7.7 Matching Logic Proof System

A sound and complete proof system of matching logic is firstly proposed in [23] and reviewed in Section 2.4. The proof system inductively defines a *provability* relation of patterns. It consists of several inference rule schemas which are of the form

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

This inductive provability relation is naturally captured in the meta-theory K using a predicate symbol

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

The main task of this subsection is to define axioms for `#provable` in correspondence to the inference rule schemas of the proof system of matching logic. Roughly speaking, the intuition is that for any inference rule schema of matching logic

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

we define the following axiom in the meta-theory K

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

in which $\llbracket \varphi_1 \rrbracket, \ldots, \llbracket \varphi_n \rrbracket, \llbracket \psi \rrbracket$ are meta-representations of $\varphi_1, \ldots, \varphi_n, \psi$ in the meta-theory. This is just a rough but good enough intuition, because in practice one also needs to consider the so-called *wellformedness premises*, which make the axioms in the meta-theory even longer. Readers should not worry about the details for now as they will be clarified in later sections. Instead, readers

should try to understand the big picture here. What we are doing is to *encode or represent each inference rule schema of matching logic by an axiom in the meta-theory K*. While each inference rule schema has infinitely many instances, we manage to *intensionally* define that infinitely many instances using one axiom of the meta-theory.

Reader should get more familiar with the double bracket $[\![\_]\!]$, also known as the semantics bracket, when we introduce the Kore language in Section 9. For now it is sufficient to think of it as purely a notation that denotes the meta-representation of something.

In the following, we will define all the axioms for the predicate symbol `#provable` in correspondence to all the inference rule schemas of matching logic defined in Section 2.4. We recall our conventions for variable names. We are using

x	as a shorthand of x:`#String`	which is a variable of sort `#String`
y	as a shorthand of y:`#String`	which is a variable of sort `#String`
z	as a shorthand of z:`#String`	which is a variable of sort `#String`
s	as a shorthand of s:`#Sort`	which is a variable of sort `#Sort`
σ	as a shorthand of σ:`#Symbol`	which is a variable of sort `#Symbol`
v	as a shorthand of v:`#Variable`	which is a variable of sort `#Variable`
u	as a shorthand of u:`#Variable`	which is a variable of sort `#Variable`
φ	as a shorthand of φ:`#Pattern`	which is a variable of sort `#Pattern`
ψ	as a shorthand of ψ:`#Pattern`	which is a variable of sort `#Pattern`
L	as a shorthand of L:`#PatternList`	which is a variable of sort `#PatternList`
R	as a shorthand of R:`#PatternList`	which is a variable of sort `#PatternList`

Finally, notice that all the axioms are axiom schemas, with $\#s \in S_K$ being any sort of the meta-theory.

Propositional Logic Inference Rules.

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

$$\forall\varphi\forall\psi\forall s \left(\underbrace{\texttt{\#wellFormed}\{\#s\}(\overline{\varphi \to_s (\psi \to_s \varphi)})}_{\text{if the meta-representation is wellformed}} \to \underbrace{\texttt{\#provable}\{\#s\}(\overline{\varphi \to_s (\psi \to_s \varphi)})}_{\text{then it is provable}} \right)$$

Let us pause here and clarify some points. Firstly, the lengthy pattern $\texttt{\#wellFormed}\{\#s\}(\overline{\varphi \to_s (\psi \to_s \varphi)})$ is called a *welformedness premise*. They are necessary, and they are crucial in establishing the faithfulness theorem (Section 11). However, they are also lengthy and boring, and they are often understood and taken for granted. Therefore, just for simplicity, we will omit writing the well-formedness premises from now on. We also omit the explicitly universal quantifiers hanging in front of the axioms. Secondly, we use the Greek letters "φ" and "ψ" both in the inference rule schema

$$\vdash \varphi \to (\psi \to \varphi) \qquad // \ \varphi \text{ and } \psi \text{ are meta-variables of patterns}$$

and in the meta-theory axiom

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

When we use them in the inference rule schema, they are mathematical variables or the so-called meta-variables for wellformed matching logic patterns. When we use then in the meta-theory axiom, they are `#Pattern` variables and thus can be quantified. This abuse use of notations is intended because *meta-variables in object theories are variables in the meta-theory.*

Let us move on and define the axioms for the other two propositional logic inference rules, in which we omit wellformedness premises and hanging universal quantifiers.

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

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

The Modus Ponens inference rule has two premises. It can be defined as an axiom in the meta-theory, too.

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

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

From now on, we will quickly list all the inference rule schemas of matching logic and their corresponding axioms in the meta-theory, and only add explanations by need.

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.

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

(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.

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

(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 $\vdash v \in \varphi$.

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

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

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

(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 $\sigma(\dots \varphi_i \dots)$.

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

The Axiom Rule.

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

$$\underbrace{\texttt{\#axiomDeclared}\{\#s\}(\varphi)}_{\text{if } \varphi \text{ is an axiom in the current theory}} \quad \rightarrow \quad \underbrace{\texttt{\#provable}\{\#s\}(\varphi)}_{\text{then } \varphi \text{ is provable in the current theory}}$$

7.8 A Summary of the Meta-Theory

In this subsection, we give a summary of the meta-theory $K = (S_K, \Sigma_K, A_K)$. We list all its sorts and symbols, together with their meaning.

7.8.1 Sorts in Meta-Theory

There are in total 10 sorts in the meta-theory.

Sort	Meaning
`#Char`	Characters
`#CharList` or `#String`	Strings
`#Sort`	Sorts
`#SortList`	Cons-lists of sorts
`#Symbol`	Symbols
`#SymbolList`	Cons-lists of symbols
`#Variable`	Variables
`#VariableList`	Cons-lists of variables
`#Pattern`	Patterns
`#PatternList`	Cons-lists of patterns

7.8.2 Symbols in Meta-Theory

Symbol or Symbol Schema	Meaning
$\lceil \_ \rceil^{\#s_2}_{\#s_1}$	$10 \times 10 = 100$ definedness symbols in the meta-theory, defined for each $\#s_1, \#s_2 \in S_K$
'a', 'b', ...	65 individual characters
`#nilCharList` or `#epsilon`	Empty string
`#consCharList`	Concatenate a character to a string
`#appendKCharList` or `#concat`	Concatenate two strings
`#sort`	Construct sorts
`#nilSortList`	Empty sort list
`#consSortList`	Construct sort lists
`#appendSortList`	Append two sort lists
`#inSortList{`$\#s$`}`	Check whether a sort belongs to a sort list
`#deleteSortList`	Delete a sort from a sort list
`#symbol`	Construct symbols
`#getArgumentSorts`	Get the argument sorts of a symbol
`#getReturnSort`	Get the return sort of a symbol
`#'ceil`	Construct meta-representations of definedness symbols
`#nilSymbolList`	Empty symbol list
`#consSymbolList`	Construct symbol lists
`#appendSymbolList`	Append two symbol lists
`#inSymbolList{`$\#s$`}`	Check whether a symbol belongs to a symbol list
`#deleteSymbolList`	Delete a symbol from a symbol list
`#variable`	Construct variables $x{:}s$
`#nilVariableList`	Empty variable list
`#consVariableList`	Construct variable lists
`#inVariableList{`$\#s$`}`	Check whether a variable belongs to a variable list
`#appendVariableList`	Append two variable lists
`#deleteVariableList`	Delete a variable from a variable list
`#variableAsPattern`	Injection function from `#Variable` to `#Pattern`
`#variablePattern`	Construct variable patterns of the form $x{:}s$
`#application`	Construct patterns of the form $\sigma(\varphi_1, \ldots, \varphi_n)$
`#\and`	Construct patterns of the form $\varphi_1 \wedge_s \varphi_2$

Symbol or Symbol Schema	Meaning
`#\not`	Construct patterns of the form $\neg_s \varphi$
`#\exists`	Construct patterns of the form $\exists_s x : s'.\varphi$
`#\or`	Construct patterns of the form $\varphi_1 \vee_s \varphi_2$
`#\implies`	Construct patterns of the form $\varphi_1 \rightarrow_s \varphi_s$
`#\iff`	Construct patterns of the form $\varphi_1 \leftrightarrow_s \varphi_s$
`#\forall`	Construct patterns of the form $\forall_s x : s'.\varphi$
`#\ceil`	Construct patterns of the form $\lceil \varphi \rceil_s^{s'}$
`#\floor`	Construct patterns of the form $\lfloor \varphi \rfloor_s^{s'}$
`#\equals`	Construct patterns of the form $\varphi_1 =_s^{s'} \varphi_2$
`#\in`	Construct patterns of the form $x : s \in_s^{s'} \varphi$
`#\top`	Construct patterns of the form $\top_s^{s'}$
`#\bottom`	Construct patterns of the form $\bot_s^{s'}$
`#nilPatternList`	Empty pattern list
`#consPatternList`	Construct pattern lists
`#appendPatternList`	Append two pattern lists
`#inPatternList{`*#s*`}`	Check whether a pattern belongs to a pattern list
`#deletePatternList`	Delete a pattern from a pattern list
`#getFV`	Get free variables in a pattern
`#getFVFromPatterns`	Get all free variables in a list of patterns
`#occursFree{`*#s*`}`	Check whether a variable occurs free in a pattern
`#freshName`	Generate a fresh variable name w.r.t. a list of patterns
`#substitute`	Substitute a variable for a pattern
`#substitutePatterns`	Substitute a variabel for a list of patterns
`#sortDeclared{`*#s*`}`	Check whether a sort is declared in the current theory
`#sortsDeclared{`*#s*`}`	Check whether a list of sorts are declared in the current theory
`#symbolDeclared{`*#s*`}`	Check whether a symbol is declared in the current theory
`#axiomDeclared{`*#s*`}`	Check whether an axiom is declared in the current theory
`#wellFormed{`*#s*`}`	Check whether a pattern is wellformed in the current theory
`#wellFormedPatterns{`*#s*`}`	Check whether a list of patterns are wellformed in the current theory
`#getSort`	Get the sort of a pattern in the current theory; Returns $\bot_{\texttt{\#Sort}}$ if the pattern is not wellformed
`#getSortsFromPatterns`	Get the sorts from a list of patterns; Returns $\bot_{\texttt{\#SortList}}$ if any pattern is not wellformed
`#provable{`*#s*`}`	Check whether a pattern is provable in the current theory

8 Representing Theories in the Meta-Theory

In Section 6, we showed the reason why we need a meta-theory and its role in defining the semantics of $\mathbb{K}$. In particular, we introduced the reflection property

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

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

However, we did not show what the meta-representations $[\![T]\!]$ and $[\![\varphi]\!]$ look like. In this section, we will use an example theory to show how to obtain the meta-representations $[\![T]\!]$ and $[\![\varphi]\!]$. This section also serves as a warming-up section of Section 9, where we define the Kore language.

8.1 A Running Example

We use a simple theory $T_L = (S_L, \Sigma_L, A_L)$ as our running example. The theory T_L has a sort Nat and a sort schema $List\{s\}$ for any sort $s \in S_L$. In other words, T_L has the following infinitely many sorts:

$$Nat, List\{Nat\}, List\{List\{Nat\}\}, List\{List\{List\{Nat\}\}\}, \ldots$$

For sort Nat, two constructors and a plus function are defined

$$\begin{array}{ll} O\colon \to Nat & \text{// the constant zero} \\ S\colon Nat \to Nat & \text{// the successor function} \\ \_ + \_\colon Nat \times Nat \to Nat & \text{// the plus function} \end{array}$$

For sort $List\{s\}$, $s \in S_L$, two constructors and an append function are defined

$$\begin{array}{ll} \epsilon_s\colon \to List\{s\} & \text{// the empty list} \\ \_ ::_s \_\colon s \times List\{s\} \to List\{s\} & \text{// the cons function} \\ \_@_s\_\colon List\{s\} \times List\{s\} \to List\{s\} & \text{// the append function} \end{array}$$

The theory T_L also gets some axioms. For example, the following are two axiom (schemas) in T_L.

$$x\colon Nat + O =^{s'}_{Nat} x\colon Nat$$

$$(x\colon s ::_s L_0\colon List\{s\})@_s L\colon List\{s\} =^{s'}_{List\{s\}} x\colon s ::_s (L_0\colon List\{s\}@_s L\colon List\{s\})$$

// for any sorts s, s' in T_L

8.2 Naming Functions

We use mathematical notations to define T_L, such as special letters (+,@, ::), mixfix operators ($\_ + \_$, $\_ :: \_$, $\_@\_$), and 2-dimensional representations such as subscripts ($\_@_s\_$). In order to represent T_L in the meta-theory K, we should represent those mathematical notations in a unified form using *strings*. This process is called *naming*. A naming function associates any "artifact" or "thing" in T_L with a `#String` pattern as its name. For instance, the following is a possible naming of T_L

Things in T_L	**Names as `#String` Patterns**
Non-parametric sort Nat	"`Nat`"
Parametric sort $List$	"`List`"
Non-parametric constant O	"`zero`"
Non-parametric function S	"`succ`"
Non-parametric function $\_ + \_$	"`plus`"
Parametric constant ϵ	"`nil`"
Parametric function $\_ :: \_$	"`cons`"
Parametric function $\_@\_$	"`append`"
Variable Name x	"`x`"
Variable Name L_0	"`L0`"
Variable Name L	"`L`"
...	...

Table 2: An Example Naming of T_L

Notice that we do not bother to name the meta-variable *#s* that is used as sort parameters in defining the T_L theory, because, strictly speaking, meta-variables do not belong to the T_L theory, they are rather a tool or a mechanism from the outside that help defining T_L.

There is, of course, more than one possible naming of T_L. As long as different "things" get different names, it does not matter what naming function we choose. This is known as the *principle of unique names*. In the following, we fix T_L's naming to be the one in Table 2.

8.3 Representing Sorts

Recall that in Section 7.3, we defined the constructor

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

to construct meta-representations of sorts. In T_L, the non-parametric sort *Nat* has name "`Nat`", so its meta-representation is the following `#Sort` pattern

$$\texttt{\#sort}(\ \underbrace{\text{“}\texttt{Nat}\text{”}}_{\text{name of } Nat}\ , \underbrace{\texttt{\#nilSortList}}_{\text{no sort parameter}})$$

The parametric sort *List*{*Nat*} has one sort parameter *Nat*, so its meta-representation is

$$\texttt{\#sort}(\ \underbrace{\text{“}\texttt{List}\text{”}}_{\text{name of } List}\ , \underbrace{\texttt{\#consSortList}(\texttt{\#sort}(\text{“}\texttt{Nat}\text{”}, \texttt{\#nilSortList}), \texttt{\#nilSortList})}_{\text{the singleton list } Nat \text{ of sort parameters}})$$

or using our abbreviation notation for list expressions (Notation 7):

$$\texttt{\#sort}(\text{“}\texttt{List}\text{”}, \underbrace{(\texttt{\#sort}(\text{“}\texttt{Nat}\text{”}, \texttt{\#nilSortList}))}_{\text{the singleton list } Nat \text{ of sort parameters}})$$

The parametric sort *List*{*List*{*Nat*}} has a sort parameter *List*{*Nat*}, so its meta-representation is

$$\texttt{\#sort}(\text{“}\texttt{List}\text{”}, \underbrace{(\texttt{\#sort}(\text{“}\texttt{List}\text{”}, (\texttt{\#sort}(\text{“}\texttt{Nat}\text{”}, \texttt{\#nilSortList}))))}_{\text{the singleton list } List\{Nat\} \text{ of sort parameters}})$$

Even though we use abbreviation notations for list expressions, the meta-representations can still be too lengthy to write and read. Therefore, we define some helper functions in $[\![T_L]\!]$ that allow us to write meta-representations more compactly

$$\begin{array}{ll} \texttt{\#`Nat}\colon \to \texttt{\#Sort} & \text{// } \textit{Nat} \text{ needs no sort parameter} \\ \texttt{\#`List}\colon \texttt{\#Sort} \to \texttt{\#Sort} & \text{// } \textit{List} \text{ needs one sort parameter} \end{array}$$

They are *not* constructors of `#Sort` but just helper functions, defined by the following two axioms

$$\texttt{\#`Nat} = \texttt{\#sort}(\text{“}\texttt{Nat}\text{”}, \texttt{\#nilSortList})$$
$$\forall s\colon\texttt{\#Sort}.\ (\texttt{\#`List}(s\colon\texttt{\#Sort}) = \texttt{\#sort}(\text{“}\texttt{List}\text{”}, \texttt{\#consSortList}(s\colon\texttt{\#Sort}, \texttt{\#nilSortList})))$$

Remember that s`:#Sort` is a variable of `#Sort`. With these two helper functions, we can write much shorter meta-representations:

Sorts in T_L	**Meta-Representations in K**
Nat	`#‘Nat`
$List\{Nat\}$	`#‘List(#‘Nat)`
$List\{List\{Nat\}\}$	`#‘List(#‘List(#‘Nat))`
...	...

Notice how similar the sorts in T_L and their meta-representations in K are.

Representing Sort Definedness. The theory T_L only defines sorts Nat, $List\{Nat\}$, $List\{List\{Nat\}\}$, ... and not any other sorts. This fact is important and should be represented in $[\![T_L]\!]$ using the predicate symbol `#sortDeclared{}` (see Section 7.6).

The theory T_L defines sort Nat, so we add the axiom schema to $[\![T_L]\!]$

$$\texttt{\#sortDeclared}\{\#s\}(\texttt{\#‘Nat}) \qquad \text{for any } \#s \in S_K$$

Notice that adding axiom schemas to $[\![T_L]\!]$ does not make $[\![T_L]\!]$ an infinite artifact, because there are only finitely many sorts in S_K.

Similarly, T_L defines sort $List\{s\}$ for any sort s that it defines, so we add the next axiom schema to T_L

$$\forall s\texttt{:\#Sort}.\left(\underbrace{\texttt{\#sortDeclared}\{\#s\}(s\texttt{:\#Sort})}_{\text{if } s \text{ is defined}} \rightarrow \underbrace{\texttt{\#sortDeclared}\{\#s\}(\texttt{\#‘List}(s\texttt{:\#Sort}))}_{\text{then } List\{s\} \text{ is defined}}\right)$$

Notice how meta-variable s is represented by a variable `s:#Sort` in the meta-theory.

We should explain how such axioms in $[\![T_L]\!]$ capture the least fixed point semantics of sort schemas.

8.4 Representing Symbols

Recall that we defined a constructor

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

to construct meta-representations of symbols. In T_L, the non-parametric symbol

$$O\colon \rightarrow Nat$$

takes no argument and has return sort Nat. Thus its meta-representation is

```
#symbol(“zero”,          // name of O: → Nat
        #nilSortList,    // no sort parameter
        #nilSortList,    // no argument sort
        #‘Nat)           // return sort is Nat
```

The non-parametric symbol

$$S\colon Nat \rightarrow Nat$$

takes an argument of sort *Nat* and has return sort *Nat*. Its meta-representation (using our abbreviation notation for list expressions) is

```
#symbol("succ",          // name of S
        #nilSortList,    // no sort parameter
        (#'Nat),         // one argument sort Nat
        #'Nat)           // return sort is Nat
```

The non-parametric symbol

$$\_ + \_ \colon Nat \times Nat \to Nat$$

takes two arguments of sort *Nat*. Its meta-representation is

```
#symbol("plus",          // name of _ + _
        #nilSortList,    // no sort parameter
        (#'Nat,#'Nat),   // two argument sorts Nat, Nat
        #'Nat)           // return sort is Nat
```

Now let us look at parametric symbols. The symbol for the empty list of natural numbers

$$\epsilon_{Nat} \colon \ \to List\{Nat\}$$

has a sort parameter *Nat*, no argument sort, and a return sort $List\{Nat\}$. Its meta-representation is

```
#symbol("nil",           // name of ε
        (#'Nat),         // one sort parameter Nat
        #nilSortList,    // no argument sort
        #'List(#'Nat))   // return sort is List{Nat}
```

Similarly, the symbol for the empty list of lists of numbers

$$\epsilon_{List\{Nat\}} \colon \ \to List\{List\{Nat\}\}$$

has a sort parameter $List\{Nat\}$, no argument sort, and a return sort $List\{List\{Nat\}\}$. Its meta-representation is

```
#symbol("nil",                  // name of ε
        (#'List(#'Nat)),        // one sort parameter List{Nat}
        #nilSortList,           // no argument sort
        #'List(#'List(#'Nat)))  // return sort is List{List{Nat}}
```

The symbol

$$\_ ::_{Nat} \_ \colon Nat \times List\{Nat\} \to List\{Nat\}$$

takes two arguments of sorts *Nat* and *List*{*Nat*} respectively. Its meta-representation is

$$\begin{array}{ll}
\texttt{\#symbol}(\text{“}\texttt{cons}\text{”}, & \text{// name of } _ :: _ \\
\quad (\texttt{\#`Nat}), & \text{// one sort parameter } \mathit{Nat} \\
\quad (\texttt{\#`Nat}, \texttt{\#`List}(\texttt{\#`Nat})), & \text{// two argument sorts } \mathit{Nat}, \mathit{List}\{\mathit{Nat}\} \\
\quad \texttt{\#`List}(\texttt{\#`Nat})) & \text{// return sort is } \mathit{List}\{\mathit{Nat}\}
\end{array}$$

The symbol

$$\_@_{Nat}\_ \colon \mathit{List}\{\mathit{Nat}\} \times \mathit{List}\{\mathit{Nat}\} \to \mathit{List}\{\mathit{Nat}\}$$

takes two arguments of sort *List*{*Nat*}. Its meta-representation is

$$\begin{array}{ll}
\texttt{\#symbol}(\text{“}\texttt{append}\text{”}, & \text{// name of } _@_ \\
\quad (\texttt{\#`Nat}), & \text{// one sort parameter } \mathit{Nat} \\
\quad (\texttt{\#`List}(\texttt{\#`Nat}), \texttt{\#`List}(\texttt{\#`Nat})), & \text{// two argument sorts } \mathit{List}\{\mathit{Nat}\}, \mathit{List}\{\mathit{Nat}\} \\
\quad \texttt{\#`List}(\texttt{\#`Nat})) & \text{// return sort is } \mathit{List}\{\mathit{Nat}\}
\end{array}$$

Representing Symbol Definedness. The theory T_L defines non-parametric symbols $O, S, \_+\_$ and parametric symbols $\epsilon_s, \_ ::_s \_, \_@_s\_$ for any sort s in T_L, and not any other symbols. This fact is represented using the predicate symbol `#symbolDeclared`{}. For non-parametric symbols, we add the next axioms to $[\![T_L]\!]$

$$\texttt{\#symbolDeclared}\{\#s\}(\underbrace{\texttt{\#symbol}(\text{“}\texttt{zero}\text{”}, \texttt{\#nilSortList}, \texttt{\#nilSortList}, \texttt{\#`Nat})}_{\text{the meta-representation of } O\colon \to \mathit{Nat}})$$

$$\texttt{\#symbolDeclared}\{\#s\}(\underbrace{\texttt{\#symbol}(\text{“}\texttt{succ}\text{”}, \texttt{\#nilSortList}, (\texttt{\#`Nat}), \texttt{\#`Nat})}_{\text{the meta-representation of } S\colon \mathit{Nat} \to \mathit{Nat}})$$

$$\texttt{\#symbolDeclared}\{\#s\}(\underbrace{\texttt{\#symbol}(\text{“}\texttt{plus}\text{”}, \texttt{\#nilSortList}, (\texttt{\#`Nat}, \texttt{\#`Nat}), \texttt{\#`Nat})}_{\text{the meta-representation of } \_ + \_ \colon \mathit{Nat} \times \mathit{Nat} \to \mathit{Nat}})$$

For parametric symbols, we add the next axioms to $[\![T_L]\!]$

$$\forall s. \left(\underbrace{\texttt{\#sortDeclared}\{\#s\}(s)}_{\text{if } s \text{ is defined}} \to \underbrace{\texttt{\#symbolDeclared}\{\#s\}(\underbrace{\texttt{\#symbol}(\text{“}\texttt{nil}\text{”}, (s), \texttt{\#nilSortList}, \texttt{\#`List}(s)))}_{\text{the meta-representation of } \epsilon_s \colon \to \mathit{List}\{s\}}}_{\text{then } \epsilon_s \text{ is defined}} \right)$$

$$\forall s. \left(\texttt{\#sortDeclared}\{\#s\}(s) \to \texttt{\#symbolDeclared}\{\#s\}(\underbrace{\texttt{\#symbol}(\text{“}\texttt{cons}\text{”}, (s), (s, \texttt{\#`List}(s)), \texttt{\#`List}(s)))}_{\text{meta-representation of } \_ ::_s \_ \colon s \times \mathit{List}\{s\} \to \mathit{List}\{s\}}} \right)$$

$$\forall s. \left(\texttt{\#sortDeclared}\{\#s\}(s) \to \texttt{\#symbolDeclared}\{\#s\}(\underbrace{\texttt{\#symbol}(\text{“}\texttt{append}\text{”}, (s), (\texttt{\#`List}(s), \texttt{\#`List}(s)), \texttt{\#`List}(s)))}_{\text{meta-representation of } \_@_s\_ \colon \mathit{List}\{s\} \times \mathit{List}\{s\} \to \mathit{List}\{s\}}} \right)$$

Notice we abbreviate s for s`:#Sort` as we always do.

8.5 Representing Matching Logic Patterns and Axioms

The theory T_L defines many axioms. Two of them are

$$x : Nat + O =^{s'}_{Nat} x : Nat$$
$$(x : s ::_s L_0 : List\{s\})@_s L : List\{s\} =^{s'}_{List\{s\}} x : s ::_s (L_0 : List\{s\}@_s L : List\{s\})$$
$$// \text{ for any sorts } s, s' \text{ in } T_L$$

How to represent them in the meta-theory? The answer is it is similar to the way we represent sorts and symbols. We firstly obtain meta-representations of those axioms, and use the predicate symbol `#axiomDeclared` to say they are axioms of T_L.

8.5.1 Meta-Representations of Patterns

We start from the basic. Recall that we defined the constructor

$$\texttt{\#variable}\colon \underbrace{\texttt{\#String}}_{\text{variable name}} \times \underbrace{\texttt{\#Sort}}_{\text{variable sort}} \to \texttt{\#Variable}$$

to construct meta-representations of variables. Consider a variable, say $x : Nat$, which has a variable name x and a sort *Nat*. Thus its meta-representation is

$$\texttt{\#variable}(\underbrace{\text{“x”}}_{\text{the name of } x}, \underbrace{\texttt{\#`Nat}}_{\text{the meta-representation of } Nat})$$

Notice that what we get here is a `#Variable` pattern, because `#variable` is a constructor of `#Variable`. To obtain the meta-representation of $x : Nat$ as a `#Pattern` pattern, we write (see Section 7.5):

$$\texttt{\#variablePattern}(\text{“x”}, \texttt{\#`Nat})$$

The meta-representations of patterns constructed using (basic and derived) logic connectives (e.g. $\wedge_s$, $\neg_s$, $\exists_s$, $=^{s_2}_{s_1}$) are constructed with the corresponding constructors (e.g., `#\and`, `#\not`, `#\exists`, `#\equals`). We now focus on getting the meta-representations of patterns which are of the form

$$\sigma(\varphi_1, \ldots, \varphi_n)$$

Recall that we defined a corresponding constructor

$$\texttt{\#application}\colon \texttt{\#Symbol} \times \texttt{\#PatternList} \to \texttt{\#Pattern}$$

Let us first consider the simplest case—a constant symbol applied to an empty list of patterns. Take the constant symbol $O\colon \to Nat$ as an example. The pattern $O()$, often abbreviated as just O, has meta-representation

$$\texttt{\#application}(\underbrace{\texttt{\#symbol}(\text{“zero”}, \texttt{\#nilSortList}, \texttt{\#nilSortList}, \texttt{\#`Nat})}_{\text{the constant symbol } O\colon \to Nat}, \underbrace{\texttt{\#nilPatternList}}_{\text{applied to empty pattern list}})$$

To help us write this lengthy pattern, we define a helper function in $[\![T_L]\!]$

$$\texttt{\#`zero}\colon \to \texttt{\#Pattern}$$

and define an axiom

$$\#\text{`}\texttt{zero} = \underbrace{\texttt{\#application}(\texttt{\#symbol}(\text{“}\texttt{zero}\text{”}, \texttt{\#nilSortList}, \texttt{\#nilSortList}, \#\text{`}\texttt{Nat}), \texttt{\#nilPatternList})}_{\text{the meta-representation of pattern } O}$$

With this helper function defined, the meta-representation of the pattern O becomes simply $\#\text{`}\texttt{zero}$. Now consider $S(O)$, in which the symbol S is applied to a singleton list O of patterns. Its meta-representation is

$$\texttt{\#application}(\underbrace{\texttt{\#symbol}(\text{“}\texttt{succ}\text{”}, \texttt{\#nilSortList}, (\#\text{`}\texttt{Nat}), \#\text{`}\texttt{Nat})}_{\text{the symbol } S\colon Nat \to Nat}, \quad \underbrace{(\#\text{`}\texttt{zero})}_{\text{applied to pattern } O} \quad)$$

Similarly, we can define a helper function

$$\#\text{`}\texttt{succ}\colon \texttt{\#Pattern} \to \texttt{\#Pattern}$$

and define an axiom

$$\forall\varphi.\left(\#\text{`}\texttt{succ}(\varphi) = \underbrace{\texttt{\#application}(\texttt{\#symbol}(\text{“}\texttt{succ}\text{”}, \texttt{\#nilSortList}, (\#\text{`}\texttt{Nat}), \#\text{`}\texttt{Nat}), (\varphi))}_{\text{the meta-representation of pattern } S(\varphi)}\right)$$

This allows us to write the meta-representation of $S(O)$ as just $\#\text{`}\texttt{succ}(\#\text{`}\texttt{zero})$.

Let us do the same thing to the plus function $\_ + \_$. We define a helper function

$$\#\text{`}\texttt{plus}\colon \texttt{\#Pattern} \times \texttt{\#Pattern} \to \texttt{\#Pattern}$$

and define an axiom

$$\forall\varphi_1\forall\varphi_2.\left(\#\text{`}\texttt{plus}(\varphi_1, \varphi_2) = \underbrace{\texttt{\#application}(\underbrace{\texttt{\#symbol}(\text{“}\texttt{plus}\text{”}, \texttt{\#nilSortList}, (\#\text{`}\texttt{Nat}, \#\text{`}\texttt{Nat}), \#\text{`}\texttt{Nat})}_{\text{the symbol } \_ + \_\colon Nat \times Nat \to Nat}, \underbrace{(\varphi_1, \varphi_2)}_{\text{two patterns}})}_{\text{the meta-representation of pattern } \varphi_1 + \varphi_2}\right)$$

These helper functions in $[\![T_L]\!]$ allows us to write compact meta-representations of patterns. For example, the meta-representation of $S(O) + S(S(O))$ is

$$\#\text{`}\texttt{plus}(\#\text{`}\texttt{succ}(\#\text{`}\texttt{zero}), \#\text{`}\texttt{succ}(\#\text{`}\texttt{succ}(\#\text{`}\texttt{zero})))$$

Now let us move on and consider parametric cases. The simplest case is the parametric empty list ϵ_s. Its meta-representation is

$$\texttt{\#application}(\underbrace{\texttt{\#symbol}(\text{“}\texttt{nil}\text{”}, (s), \texttt{\#nilSortList}, (\#\text{`}\texttt{List}(s)))}_{\text{the symbol } \epsilon_s\colon \to List\{s\}}, \underbrace{\texttt{\#nilPatternList}}_{\text{empty pattern list}})$$

Let us also define for ϵ_s a helper function. It needs a sort parameter s, so its helper function takes a `#Sort` pattern as argument

$$\#\text{`}\texttt{nil}\colon \texttt{\#Sort} \to \texttt{\#Pattern}$$

It has the axiom

$$\forall s.\left(\#\text{`}\texttt{nil}(s) = \underbrace{\texttt{\#application}(\texttt{\#symbol}(\text{“}\texttt{nil}\text{”}, (s), \texttt{\#nilSortList}, (\#\text{`}\texttt{List}(s))), \texttt{\#nilPatternList})}_{\text{meta-representation of pattern } \epsilon_s}\right)$$

Using this helper function, we can easily construct meta-representations for the patterns ϵ_s for any s, as shown in the next table

Patterns in T_L	**Meta-Representations in K**
ϵ_{Nat}	#`nil(#`Nat)
$\epsilon_{List\{Nat\}}$	#`nil(#`List(#`Nat))
$\epsilon_{List\{List\{Nat\}\}}$	#`nil(#`List(#`List(#`Nat)))
...	...

The parametric symbol $\_ ::_s \_$ needs a sort parameter s and it takes two patterns as arguments. Let us define its helper function

$$\#\text{`}\texttt{cons}: \underbrace{\texttt{\#Sort}}_{\text{a sort parameter}} \times \underbrace{\texttt{\#Pattern} \times \texttt{\#Pattern}}_{\text{two patterns}} \to \texttt{\#Pattern}$$

and define the axiom

$$\forall s \forall \varphi_1 \forall \varphi_2.\left(\#\text{`}\texttt{cons}(s, \varphi_1, \varphi_2) = \underbrace{\texttt{\#application}(\texttt{\#symbol}(\text{“}\texttt{cons}\text{”}, (s), (s, \#\text{`}\texttt{List}(s)), \#\text{`}\texttt{List}(s)), (\varphi_1, \varphi_2))}_{\text{meta-representation of pattern } \varphi_1 ::_s \varphi_2}\right)$$

Using this helper function, we can write the meta-representation of $O ::_{Nat} (S(O) ::_{Nat} \epsilon_{Nat})$ as

$$\#\text{`}\texttt{cons}(\#\text{`}\texttt{Nat}, \#\text{`}\texttt{zero}, \#\text{`}\texttt{cons}(\#\text{`}\texttt{Nat}, \#\text{`}\texttt{succ}(\#\text{`}\texttt{zero}), \#\text{`}\texttt{nil}(\#\text{`}\texttt{Nat})))$$

Similarly, the parametric symbol $\_@_s\_$ has its helper function

$$\#\text{`}\texttt{append}: \texttt{\#Sort} \times \texttt{\#Pattern} \times \texttt{\#Pattern} \to \texttt{\#Pattern}$$

We left the definition of its axiom as an exercise to readers.

8.5.2 Representing Axioms

The theory T_L defines some axioms. For example

$$x : Nat + O =^{s'}_{Nat} x : Nat$$
$$(x : s ::_s L_0 : List\{s\})@_s L : List\{s\} =^{s'}_{List\{s\}} x : s ::_s (L_0 : List\{s\}@_s L : List\{s\})$$
// for any sorts s, s' defined in T_L

To represent those axioms and axiom schemas, we first calculate their meta-representations:

$$
\begin{array}{l}
\text{// Let } \Psi_1 \text{ be} \\
\quad \texttt{\#\textbackslash equals}(\texttt{\#`Nat}, s', \\
\qquad\qquad \texttt{\#`plus}(\texttt{\#variablePattern}(\text{“}\texttt{x}\text{”}, \texttt{\#`Nat}), \texttt{\#`zero}), \\
\qquad\qquad \texttt{\#variablePattern}(\text{“}\texttt{x}\text{”}, \texttt{\#`Nat})) \\
\text{// Let } \Psi_2 \text{ be} \\
\quad \texttt{\#\textbackslash equals}(\texttt{\#`List}(s), s', \\
\qquad\qquad \texttt{\#`append}(s, \texttt{\#`cons}(s, \texttt{\#variablePattern}(\text{“}\texttt{x}\text{”}, s), \\
\qquad\qquad\qquad\qquad\qquad \texttt{\#variablePattern}(\text{“}\texttt{L0}\text{”}, \texttt{\#`List}(s))), \\
\qquad\qquad\qquad\qquad \texttt{\#variablePattern}(\text{“}\texttt{L}\text{”}, \texttt{\#`List}(s))), \\
\qquad\qquad \texttt{\#`cons}(s, \texttt{\#variablePattern}(\text{“}\texttt{x}\text{”}, s), \\
\qquad\qquad\qquad\qquad \texttt{\#`append}(s, \texttt{\#variablePattern}(\text{“}\texttt{L0}\text{”}, \texttt{\#`List}(s)), \\
\qquad\qquad\qquad\qquad\qquad\qquad \texttt{\#variablePattern}(\text{“}\texttt{L}\text{”}, \texttt{\#`List}(s)))))
\end{array}
$$

Then we can define the following axiom schemas in $[\![T_L]\!]$

$$\forall s'.(\underbrace{\texttt{\#sortDeclared}\{\#s\}(s')}_{\text{if } s' \text{ is defined in } T_L} \rightarrow \underbrace{\texttt{\#axiomDeclared}\{\#s\}(\Psi_1)}_{\text{then } \Psi_1 \text{ is an axiom in } T_L})$$

$$\forall s \forall s'.(\underbrace{\texttt{\#sortDeclared}\{\#s\}(s) \wedge \texttt{\#sortDeclared}\{\#s\}(s')}_{\text{if } s, s' \text{ are defined in } T_L} \rightarrow \underbrace{\texttt{\#axiomDeclared}\{\#s\}(\Psi_2)}_{\text{then } \Psi_2 \text{ is an axiom in } T_L})$$

8.6 Faithfulness

We present the faithfulness theorem of the meta-theory, whose proof is in Section 11.

Theorem 10 (Faithfulness Theorem)**.** *Given a matching logic theory T and a naming function. Let $[\![T]\!]$ be the meta-theory instantiated with T, and φ be any pattern in T,*

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

Theorem 10 is an important theorem both in theory and in practice. It relates the provability relation of infinite theories (“ $\vdash$ ”) with the provability relation of finite theories (“ $\vdash_{\text{fin}}$ ”), the only one that could ever be implemented.

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 again of the reflection property that we firstly introduced in Section 6

$$T \vdash \varphi \quad \text{iff} \quad K \cup [\![T]\!] \vdash_{\text{fin}} \texttt{\#provable}\{\#s\}([\![\varphi]\!])$$

Given such a reflection property, we could use the finite theory $[\![T]\!]$ *as the specification* of the possibly infinite theory T. However, the meta-theory K and the meta-representation $[\![T]\!]$ are 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 process*, which is giving Kore a semantics as a specification language (Section 9.2).

9.1 The Kore Language Syntax

This section defines a complete and concrete syntax of the Kore language, the design of which is mostly driven by the goal of relatively easy parsing without making too many compromises on human readability. The Kore language syntax is defined by BNF-style production rules. The language accepted by there rules is a superset of the Kore language. A legal Kore language source file should satisfy additional constraints such as wellformedness (see Section 9.1.8).

The Kore language is a formal specification language that is designed for the meta-theory (Section 7), which gives us all power of reasoning and proving by the faithfulness theorem (Theorem 10). The semantics of Kore is given based on the meta-theory (Section 9.2). One can also think of all sorts, symbols, and axioms of the meta-theory are ready to use to any Kore definition *as if* they are implicitly predefined. Precisely speaking, they are *built-in* to the Kore language.

9.1.1 Lexicon

A Kore language source file, also called as a Kore definition, consists of a sequence of ASCII characters. All lexical tokens defined below are limited to ASCII printable characters.

Comments. Kore supports C-style comments. Line comments start with "`//`" and block comments start with "`/*`" and end with "`*/`". Both comments and consecutive white space characters occurring outside a string literal or an identifier (both defined later) are considered *whitespace*. The only lexical function of whitespace is to break the source text into tokens.

The lexicon tokens of Kore include a few token characters (listed in Table 3) and the elements of the syntactic categories ⟨*identifier*⟩, ⟨*char*⟩, ⟨*string*⟩, ⟨*keyword*⟩, which we all defined below.

Token Character(s)	Name
`:`	Colon
`{ }`	Curly Braces
`( )`	Parenthesis
`[ ]`	Braces
`,`	Comma

Table 3: Lexicon Tokens in Kore

Identifiers. Identifiers are mostly used as symbols, sorts, and variable names. An ⟨*identifier*⟩ is either of the following two cases:

- An ⟨*object-identifier*⟩, which is a nonempty sequence of letters, digits, the character "`'`", and the character "`-`" that starts with a letter and is not a ⟨*keyword*⟩.
- A ⟨*meta-identifier*⟩, which is either the character "`#`" followed by an ⟨*object-identifier*⟩, or the string "`#‘`" followed by an ⟨*object-identifier*⟩, or one of the following:

`#\and`	`#\not`	`#\or`	`#\implies`	`#\iff`
`#\forall`	`#\exists`	`#\ceil`	`#\floor`	`#\equals`
`#\in`	`#\top`	`#\bottom`		

Identifiers are building blocks of many syntactic categories in Kore, such as ⟨*sort*⟩, ⟨*symbol*⟩, and ⟨*pattern*⟩. It is important to be able to syntactically distinguish ⟨*object-identifier*⟩ and ⟨*meta-identifier*⟩ because they will be used to represent different semantics components and we do not want to take the risk of confusing them.

Character and string literals. Kore supports C-style character and string literals. Character literals are wrapped with single quotes while string literals are wrapped with double quotes. The syntactic category of character literals is ⟨*char*⟩. We use character literals to represent patterns in the meta-theory whose sort is `#Char`. The syntactic category of string literals is ⟨*string*⟩. We use string literals to represent patterns in the meta-theory whose sort is `#String` (a.k.a., `#CharList`).

Keywords. Kore has six keywords. They belong to the syntactic category ⟨*keyword*⟩. They are used to signify the begin and end of a declaration.

Keyword	Usage
`module`	Initiate a module declaration
`endmodule`	Finalize a module declaration
`sort`	Initiate a sort declaration
`symbol`	Initiate a symbol declaration
`alias`	Initiate an alias declaration
`axiom`	Initiate an axiom declaration

9.1.2 Sorts

A sort is either a *sort variable* or a *sort constructor* applied to a list of *sort parameters*. Sorts include object-level sorts and meta-level sorts, which are syntactically different as we will see in the next grammar, so that we can tell whether a sort is object-level or meta-level in the parse time.

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

⟨*object-sort*⟩ ::= ⟨*object-sort-variable*⟩ | ⟨*object-non-variable-sort*⟩

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

⟨*object-non-variable-sort*⟩ ::= ⟨*object-sort-constructor*⟩ ‘`{`’ ⟨*object-sort-list*⟩ ‘`}`’

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

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

⟨*meta-sort*⟩ ::= ⟨*meta-sort-variable*⟩ | ⟨*meta-non-variable-sort*⟩

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

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

⟨*meta-sort-constructor*⟩ ::= '`#Char`' | '`#CharList`' | '`#String`' | '`#Sort`' | '`#SortList`'
| '`#Symbol`' | '`#SymbolList`' | '`#Variable`' | '`#VariableList`' | '`#Pattern`' | '`#PatternList`'

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

If a sort has no sort parameter, it is a *non-parametric* sort. All meta-level sorts are non-parametric sorts, as defined in the grammar. Notice that any sort that is not a sort variable is of the form $C\{s_1, \ldots, s_n\}$ for some $n \geq 0$, where C is sort constructor and $s_1, \ldots, s_n$ are sort parameters. Even though we do exclude ⟨*keyword*⟩ from ⟨*object-identifier*⟩, and thus from ⟨*object-sort-variable*⟩, we do not exclude ⟨*meta-sort-constructor*⟩ from ⟨*meta-sort-variable*⟩. As a result, one may use `#Char` as a valid meta-sort variable in Kore, even though it is in general a bad practice because it is confusing. It might appear that the policies we have here are inconsistent, but it is really not. In the following, we will see that whenever a sort variable is used, it has to be declared by putting in a list of sort variables wrapped with curly brackets (see Section 9.1.6), so that we can tell if something is a sort variable locally. As a result, we do not need to exclude any ⟨*object-sort-constructor*⟩ from ⟨*object-sort-variable*⟩, as whenever an ⟨*object-identifier*⟩ is used as sort variable, we can tell that locally from the list of declared sort variables. For the same reason, we do not exclude any ⟨*meta-sort-constructor*⟩ from ⟨*meta-sort-variable*⟩. On the other hand, it is a common practice to exclude keywords from identifiers in programming languages, and in Kore we follow the same practice.

Kore has a fixed number of meta-sorts, as illustrated in the grammar. One can think of these meta-sorts are implicitly declared. In fact, they are a part of the meta-theory that we introduced in Section 7.

9.1.3 Heads

Heads are user-definable syntactic constructors of ⟨*pattern*⟩ (see Section 9.1.4). In practice, a head is either a matching logic symbol or an alias. The only difference between symbols and aliases are their semantics. The semantics of a symbol is defined not only by user-defined axioms, but also by the inference rule (Membership Symbol) in the matching logic proof system (see Section 7.7). The semantics of an alias is defined only by user-defined axioms. Whether a head is declared as a symbol or an alias is decided in its declaration (see Section 9.1.6). If it is declared with the keyword `symbol`, then the head is a symbol. If it is declared with the keyword `alias`, then the head is an alias. Therefore the parser cannot tell if a head appearing in a pattern is a symbol or an alias *locally*. The parser either needs to remember a table of all symbols and aliases declarations and look up the table whenever it parses a head, or it simply parses the head as it is and leaves the job of looking up declaration table to other tools. In here we prefer the latter approach, because the job of looking up declaration table to decide if a head is a symbol or an alias is also part of the job of *checking wellformedness* of the definition, for which we need separate tools anyway.

Heads include ⟨*object-head*⟩ and ⟨*meta-head*⟩. They are used to construct ⟨*object-pattern*⟩ and ⟨*meta-pattern*⟩ respectively, as shown in the next grammar.

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

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

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

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

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

A head is of the form $C\{s_1, \ldots, s_n\}$ for some $n \geq 0$, where C is head constructor and $s_1, \ldots, s_n$ are sort parameters. In the case when $n = 0$, we say the head constructor C is a non-parametric head constructor and the head $C\{\}$ is a non-parametric head. Notice that an object head constructor can only take object sorts as parameters, and a meta-level head constructor can only take meta-level sorts as parameters.

9.1.4 Patterns

Apart from a few exceptions, patterns in Kore have a highly unified prefix form representations. Patterns include ⟨*object-pattern*⟩ and ⟨*meta-pattern*⟩.

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

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

⟨*object-pattern*⟩ ::=
- ⟨*object-variable*⟩
- ⟨*object-head*⟩ '`(`' ⟨*pattern-list*⟩ '`)`'
- '`\and`' '`{`' ⟨*object-sort*⟩ '`}`' '`(`' ⟨*pattern*⟩ '`,`' ⟨*pattern*⟩ '`)`'
- '`\not`' '`{`' ⟨*object-sort*⟩ '`}`' '`(`' ⟨*pattern*⟩ '`)`'
- '`\or`' '`{`' ⟨*object-sort*⟩ '`}`' '`(`' ⟨*pattern*⟩ '`,`' ⟨*pattern*⟩ '`)`'
- '`\implies`' '`{`' ⟨*object-sort*⟩ '`}`' '`(`' ⟨*pattern*⟩ '`,`' ⟨*pattern*⟩ '`)`'
- '`\iff`' '`{`' ⟨*object-sort*⟩ '`}`' '`(`' ⟨*pattern*⟩ '`,`' ⟨*pattern*⟩ '`)`'
- '`\forall`' '`{`' ⟨*object-sort*⟩ '`}`' '`(`' ⟨*object-variable*⟩ '`,`' ⟨*pattern*⟩ '`)`'
- '`\exists`' '`{`' ⟨*object-sort*⟩ '`}`' '`(`' ⟨*object-variable*⟩ '`,`' ⟨*pattern*⟩ '`)`'
- '`\ceil`' '`{`' ⟨*object-sort*⟩ '`,`' ⟨*object-sort*⟩ '`}`' '`(`' ⟨*pattern*⟩ '`)`'
- '`\floor`' '`{`' ⟨*object-sort*⟩ '`,`' ⟨*object-sort*⟩ '`}`' '`(`' ⟨*pattern*⟩ '`)`'
- '`\equals`' '`{`' ⟨*object-sort*⟩ '`,`' ⟨*object-sort*⟩ '`}`' '`(`' ⟨*pattern*⟩ '`,`' ⟨*pattern*⟩ '`)`'
- '`\in`' '`{`' ⟨*object-sort*⟩ '`,`' ⟨*object-sort*⟩ '`}`' '`(`' ⟨*pattern*⟩ '`,`' ⟨*pattern*⟩ '`)`'
- '`\top`' '`{`' ⟨*object-sort*⟩ '`}`' '`(`' '`)`'
- '`\bottom`' '`{`' ⟨*object-sort*⟩ '`}`' '`(`' '`)`'

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

⟨*meta-pattern*⟩ ::=
- ⟨*meta-variable*⟩
- ⟨*string*⟩
- ⟨*char*⟩
- ⟨*meta-head*⟩ '`(`' ⟨*pattern-list*⟩ '`)`'
- '`\and`' '`{`' ⟨*meta-sort*⟩ '`}`' '`(`' ⟨*pattern*⟩ '`,`' ⟨*pattern*⟩ '`)`'
- '`\not`' '`{`' ⟨*meta-sort*⟩ '`}`' '`(`' ⟨*pattern*⟩ '`)`'
- '`\or`' '`{`' ⟨*meta-sort*⟩ '`}`' '`(`' ⟨*pattern*⟩ '`,`' ⟨*pattern*⟩ '`)`'
- '`\implies`' '`{`' ⟨*meta-sort*⟩ '`}`' '`(`' ⟨*pattern*⟩ '`,`' ⟨*pattern*⟩ '`)`'
- '`\iff`' '`{`' ⟨*meta-sort*⟩ '`}`' '`(`' ⟨*pattern*⟩ '`,`' ⟨*pattern*⟩ '`)`'

| ‘\forall’ ‘{’ ⟨*meta-sort*⟩ ‘}’ ‘(’ ⟨*meta-variable*⟩ ‘,’ ⟨*pattern*⟩ ‘)’
| ‘\exists’ ‘{’ ⟨*meta-sort*⟩ ‘}’ ‘(’ ⟨*meta-variable*⟩ ‘,’ ⟨*pattern*⟩ ‘)’
| ‘\ceil’ ‘{’ ⟨*meta-sort*⟩ ‘,’ ⟨*meta-sort*⟩ ‘}’ ‘(’ ⟨*pattern*⟩ ‘)’
| ‘\floor’ ‘{’ ⟨*meta-sort*⟩ ‘,’ ⟨*meta-sort*⟩ ‘}’ ‘(’ ⟨*pattern*⟩ ‘)’
| ‘\equals’ ‘{’ ⟨*meta-sort*⟩ ‘,’ ⟨*meta-sort*⟩ ‘}’ ‘(’ ⟨*pattern*⟩ ‘,’ ⟨*pattern*⟩ ‘)’
| ‘\in’ ‘{’ ⟨*meta-sort*⟩ ‘,’ ⟨*meta-sort*⟩ ‘}’ ‘(’ ⟨*pattern*⟩ ‘,’ ⟨*pattern*⟩ ‘)’
| ‘\top’ ‘{’ ⟨*meta-sort*⟩ ‘}’ ‘(’ ‘)’
| ‘\bottom’ ‘{’ ⟨*meta-sort*⟩ ‘}’ ‘(’ ‘)’

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

⟨*pattern-list*⟩ ::= ϵ | ⟨*pattern*⟩ | ⟨*pattern*⟩ ‘,’ ⟨*pattern-list*⟩

Notice that we allow a mixed syntax for object-level patterns and meta-level patterns. In practice it is useful, because meta-level patterns are often lengthier and less intuitive than the corresponding object-level ones. On the other hand, we have seen there are many cases in which one has no means but to use the meta-theory and write meta-level patterns to define certain things, such as parametric sorts and side conditions (see Section 3). By allowing writing a mix of object-level patterns and meta-level patterns, the users can write patterns in the object-level as usual and only write meta-level patterns when needed.

Apart from the logic connectives that are defined in the above grammar, there are a few more connectives and symbols that are often used in program verification in Kore. These connectives and symbols include those that are defined in Section 4. The “next” symbol, denoted as $\circ\{s\}$ with a parameter s, is a unary symbol. The “rewrites” construct, denoted as $\_\Rightarrow\{s\}\_$, is a binary construct defined by $\varphi\Rightarrow\{s\}\psi \equiv \varphi \rightarrow \circ\{s\}\psi$. They have the following productions:

⟨*object-pattern*⟩ ::=
| ‘\next’ ‘{’ ⟨*object-sort*⟩ ‘}’ ‘(’ ⟨*pattern*⟩ ‘)’
| ‘\rewrites’ ‘{’ ⟨*object-sort*⟩ ‘}’ ‘(’ ⟨*pattern*⟩ ‘,’ ⟨*pattern*⟩ ‘)’

Notice that these constructs are only defined for object-level patterns. The meta-theory does not have these constructs defined, (see Section 7 for details about the meta-theory), and thus do not need a syntax for these constructs.

No `#up`, no `#down`, no problem. In Kore, we adopt a syntax for ⟨*pattern*⟩ that allows a mixed-use of both ⟨*object-pattern*⟩ and ⟨*meta-pattern*⟩. The intended semantics of such a mixed syntax, as we will define in detail in Section 9.2, is that ⟨*object-pattern*⟩ are just syntactic sugar for things at the meta-level (in fact, their abstract syntax trees), and they will all be automatically lifted to the meta-level. Compared to this implicit-level-shifting manner, an alternative approach is to use `#up` and `#down` to explicitly shift levels. However, we found this alternative approach less favorable than our approach, for the following reasons.

Firstly, it is not clear what `#up` and `#down` really are from a mathematical point of view. Therefore it is at risk to introduce them because of their lack of clear mathematical meaning: users might think of them to be what they are not. They are external to the logic and thus none of the logical proof rules or semantics applies to them. For example, one cannot do any reasoning in the argument of `#up`, because two things that are logically equivalent may have different abstract syntax trees, and thus will be taken to two different things by `#up`. Similarly for `#down`, one would have to do just ”enough” reasoning in its argument until it becomes the meta-level representation of an object pattern, which is risky and possibly lead to inconsistencies.

Secondly, `#up` and `#down` provide no real benefit for the users; on the contrary, they make them type more. For example in lambda calculus, we have the β-reduction axiom (side conditions omitted because they are not important for this discussion)

$$(\lambda x.e)e' = e[e'/x]$$

Notice that e and e' have to be meta-variables, and therefore one would write in Kore `#E:#Pattern` and `#E':#Pattern` for each of them, resp. Now, in order to plug these two meta-variables in place, we need to wrap them with an explicit `#down`, to bring them from the meta-level to the object level, and end up with something (that we do not want) as follows

```
axiom
  \equals{Exp,Exp}(
    app{}(lambda{}(X:Exp,#down(#E:#Pattern)),
         #down(#E':#Pattern)),
    #down(#substitute{}(#down(#E:#Pattern), #down(#E':#Pattern), #up(X:Exp))))
```

In fact, by having our current convention that all concrete syntax is automatically lifted, there is nothing to stop users from writing

```
   #inVariableList{}(X:S, #getFV{}(mult{}(X:S, zero{}()))
```

There is no need for `#up` and `#down` in order for users to not be exposed to the meta-level. That is, they can already do that methodologically. And backends like K-2-LLVM will still only know a few "meta-level" constructs, like `#inVariableList`, and can still only accept definitions that make no explicit use of meta-level notation.

Finally, there is the issue of checking well-sortedness and guaranteeing a sort-preservation kind of result. We know that we can screw things up massively at the meta-level. For example, when using

`#getFV`: `#Pattern` $\rightarrow$ `#VariableList`

'FV : KPattern -¿ KPattern' like this '... andBool X inSet FV(X * 0)', you are assuming that the 'KPattern' it returns is the meta-level representation of some 'Set' core pattern. But if you implement 'FV' at the meta-level in a way where it gives a meta-level representation of a 'List', or even worse, some pure meta-level 'KPatternList', then the well-sortedness is broken We were hoping that with explicit '#up' and '#down' we can prevent violations of sortedness, where we write 'andBool X inSet #downSet(FV(#up(X * 0)))' and then we can have '#down' do a sort checking at runtime this way, we can statically sort-check the core definition, modulo the claims made by '#down' which are checked at runtime, so we are sort-safe (edited) Well, this runtime check for well-sortedness of meta-stuff may or may not be a good thing in itself, but it is toally orthogonal to having '#up' and '#down' in KORE. That is, there is nothing to stop you from adding it around all meta-level patterns that occur in core-level contexts. We may add it as a kompilation option, for example. But you do not need any '#down' for that, and sort inference is super-trivial, because we do not have implicit subsorting (but only explicit injection symbols).

Xiaohong: I need to understand this example better in order to phrase this discussion for well-sortedness and runtime checking.

9.1.5 Attributes

Attributes are pieces of information, often collected by front-ends, about various aspects including pretty-printing, parsing concrete syntax of user-defined languages, evaluation and rewriting strategies, associative and commutative matching, and automatic document generation. Some attributes

contain purely syntactic information while the others contain semantics information. All semantics information contained in attributes has to have corresponding axioms explicitly declared in Kore definitions, and thus in theory we do not need to look at attributes at all in Kore. However, information contained in attributes is highly compact and easy to understand. Many semantics attributes provide useful hints for back-ends to use faster algorithms or more efficient data structures. Even though such hints are encoded as axioms in Kore definition, we do not want back-ends to re-engineer the information from those axioms. Therefore, in Kore definitions we provide syntax to specify attributes for every declaration. Back-ends that care less about proving and focus more on execution may choose to simply trust these attributes, without checking that a definition does contain axioms that justify the attributes it claims. Back-ends that care more about proving and want to have *certificates* for whatever they do need to use axioms instead of attributes to generate proof objects and prove them in matching logic provers.

Attributes are written as comma-separated patterns wrapped with brackets "`[`" and "`]`".

⟨*attribute*⟩ ::= '`[`' ⟨*pattern-list*⟩ '`]`'

9.1.6 Kore Modules and Declarations

A Kore module contains a module name and a sequence of declarations. Every declaration declares a module import, a sort, a symbol, an alias, or an axiom.

⟨*module*⟩ ::=
  | '`module`' ⟨*module-name*⟩ ⟨*declaration*⟩* '`endmodule`' ⟨*attribute*⟩

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

⟨*import-declaration*⟩ ::= '`import`' ⟨*module-name*⟩ ⟨*attribute*⟩

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

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

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

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

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

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

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

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

⟨*sort-variable-list*⟩ ::=
  | ϵ
  | ⟨*object-sort-variable*⟩
  | ⟨*meta-sort-variable*⟩
  | ⟨*object-sort-variable*⟩ '`,`' ⟨*sort-variable-list*⟩
  | ⟨*meta-sort-variable*⟩ '`,`' ⟨*sort-variable-list*⟩

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

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

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

Every declaration has a list of sort variables wrapped in curly brackets, also called as a *sort variable declaration*. In axiom declarations, the sort variable declarations follow by the keyword '`axiom`'. In sort, symbol and alias declarations, the sort variable declarations follow by the corresponding constructors that are declared. Sort variable declarations are used to help us decide *in a local manner* if an identifier is a sort variable or something else. All sort variables used in a declaration have to be declared in the sort variable declaration (see Section 9.1.8).

9.1.7 Kore Definitions

A Kore definition contains an attribute and a sequence of modules. A Kore definition contains at least a module.

⟨*definition*⟩ ::= ⟨*attribute*⟩ ⟨*module*⟩$^+$

9.1.8 Wellformedness

The language defined by the BNF grammar we just showed forms a superset of the Kore language. For a Kore definition to be valid, it has to satisfy the following common wellformedness conditions.

1. In any declaration, every sort variable used should be included in the sort variable list; The sort variable lists should contain only distinct sort variables (no duplicates);
2. In any symbol or alias declaration, the head being declared should be of the form $C\{s_1, \ldots, s_n\}$ where C is a head constructor and $s_1, \ldots, s_n$ are n distinct sort variables;
3. Every declared symbol or alias should have their argument sorts and return sorts declared;
4. In any pattern that is a head being applied to a list of argument patterns, the number and sorts of argument patterns should match with the head declaration;
5. Every sorts, symbols, and aliases should be declared to be used;

6. Every sorts, symbols, and aliases should all have unique constructor names; One cannot use the same name for a sort and a symbol, for example. Similarly, any meta-symbol cannot have the same name as a meta-sort.
7. Every sort constructor can be declared only once;

Besides the above wellformedness conditions for patterns, we have the following wellformedness conditions for module imports.

1. Two modules in a definition cannot have the same name.
2. A module can only import modules that are have been declared. This guarantees that the module import relation does not form a cycle. A module does not import itself.
3. Import declarations should only occur in the beginning of a module, preceding all other declarations.
4. Any declaration in a module M is visible in the entire module (both before and after the declaration), and in any module that imports M. This means that in a module, the order of sort, symbol, alias, and axiom declarations do not matter. However, the order of two modules do matter, because a module cannot import modules that have not declared yet.

9.2 The Kore Language Semantics

Suppose one writes a Kore definition in the source file "`definition.kore`" that specifies a matching logic theory T. It is natural to think of the theory T as the semantics of the Kore definition "`definition.kore`". However, the theory T may itself be an infinite artifact. Therefore, instead of using the theory T, we use the theory $[\![T]\!]$, known as *the meta-theory instantiated with* T, as the semantics of the Kore definition "`definition.kore`". In particular, the theory $[\![T]\!]$ is itself a finite matching logic theory that can also be specified using the Kore language, say, in the Kore definition "`#definition.kore`".

We say the semantics of a Kore definition "`definition.kore`" that defines a theory T is the Kore definition "`#definition.kore`" that defines the (finite) theory $[\![T]\!]$. In fact, we regard the Kore definition "`definition.kore`" as just syntactic sugar of the (possibly lengthy but finite) Kore definition "`#definition.kore`", whose semantics, if one insists asking, is self-explained.

The process of de-sugaring "`definition.kore`" and obtaining "`#definition.kore`" is called *lifting*. In Section 8, we have shown using a running example how to go from T to $[\![T]\!]$, which is in its essence similar to the lifting process that we will define here. The difference is that in this section, we will define the lifting process formally and algorithmically as a translation from one Kore definition to the other.

General Principle of Lifting. The general principle of the lifting process is to lift any object level target to its meta-representation and keep any meta-level target unchanged. More specifically, any object level sort/symbol/alias/pattern is going to be lifted to the meta-level pattern that represents its abstract syntax tree. Any object level sort/symbol/alias/axiom declaration is going to be lifted to the meta-level axiom that says "the object level sort/symbol/alias/axiom is declared in the current theory".

Notation 11. We use the double bracket $[\![\_]\!]$ for the lifting process and overload it for all syntactic categories. Subscripts are used (e.g., $[\![\_]\!]_{name}$, $[\![\_]\!]_{list}$, ...) when we need some auxiliary lifting processes and do not want to confuse them with the canonical one.

9.2.1 Lift Object Identifiers to String Literals

Object identifiers are lifted to string literals by wrapping them with quotation marks "`"`". This lifting process is part of the naming function we introduced in Section 8.2.

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

9.2.2 Lift Object Sort Constructors to Meta Symbols

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

$$\begin{aligned}&[\![\_]\!] : \langle \textit{object-sort-constructor} \rangle \to \langle \textit{meta-head} \rangle \\ &[\![C]\!] = \text{`\#'}\ \text{`‘'}\ C\ \text{`\{\}'} \quad \text{for any object sort constructor } C\end{aligned}$$

For example,

Object sort constructor C	Non-parametric meta-level symbol $[\![C]\!]$
`Nat`	`#‘Nat{}`
`List`	`#‘List{}`
`Map`	`#‘Map{}`

9.2.3 Lift Object Sorts and Object Sort Lists

The lifting process that lifts object sorts to their meta-representations is defined inductively as follows

$$\begin{aligned}&[\![\_]\!] : \langle \textit{object-sort} \rangle \to \langle \textit{meta-pattern} \rangle \\ &[\![s]\!] = \begin{cases} \text{`\#'}\ s\ \text{`:'}\ \text{`\#Sort\{\}'} & \text{if } s \text{ is a sort variable} \\ [\![C]\!]\ \text{`(‘'}\ [\![s_1]\!]\ \text{`,'}\ \ldots\ \text{`,'}\ [\![s_n]\!]\ \text{`)'} & \text{if } s \text{ is of the form } C\{s_1, \ldots, s_n\} \end{cases}\end{aligned}$$

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`

Given a list of object sorts $s_1, \ldots, s_n$. Sometimes we want to lift the sort list to a meta-pattern of sort `#SortList`, instead of a list of meta-patterns of sort `#Sort`. For that reason, we define the following lifting process for lists

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

We remind readers that this lifting process uses helper functions as we introduced in Section 8.3 that help us write more compact meta-representations. It is also the lifting process that we use the most often. On the other hand, the lifting process that does not use these helper functions is also needed. For example in Section 9.2.4, we will show how to generate declarations for these helper functions and their axioms. In there, we will use the lifting process that does not use helper functions. For consistency, we define this lifting process, denoted as $[\![\_]\!]_{verbose}$, in this subsection.

$$[\![\_]\!]_{verbose} : \langle \textit{object-sort} \rangle \to \langle \textit{meta-pattern} \rangle$$

$$[\![s]\!] = \begin{cases} \text{`\#'}\ s\ \text{`:'}\ \texttt{\#Sort\{\}} & \text{if } s \text{ is a sort variable} \\ \texttt{\#sort\{\}}\ \text{`('}\ [\![C]\!]_{name}\ \text{`,'}\ [\![s_1, \ldots, s_n]\!]_{verboselist}\ \text{`)'} & \text{if } s \text{ is of the form } C\{s_1, \ldots, s_n\} \end{cases}$$

In the definition, $[\![\_]\!]_{verboselist}$ is the lifting process for lists that does not use helper functions. It is inductively defined as

$$[\![\_]\!]_{verboselist} : \langle \textit{object-sort-list} \rangle \to \langle \textit{meta-pattern} \rangle$$

$$[\![\alpha]\!]_{verboselist} = \begin{cases} \texttt{\#nilSortList\{\}} & \text{if } \alpha \text{ is empty} \\ \texttt{\#consSortList\{\}}\ \text{`('}\ [\![s]\!]_{verbose}\ \text{`,'}\ [\![\beta]\!]_{verboselist}\ \text{`)'} & \text{if } \alpha = s, \beta \end{cases}$$

9.2.4 Lift Sort Declarations

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

$$[\![\_]\!] : \langle \textit{sort-declaration} \rangle \to \langle \textit{declaration} \rangle \times \langle \textit{declaration} \rangle \times \langle \textit{declaration} \rangle$$

Consider the sort declaration

$$\texttt{sort}\ C\{s_1, \ldots, s_n\}\ \texttt{[}\textit{att}\texttt{]}$$

where $s_1, \ldots, s_n$ are sort variables, and C is a sort constructor, and att is an attribute. This sort declaration is lifted to the following three declarations.

The first declaration is a symbol declaration for the corresponding helper function $[\![C]\!]$ for the sort constructor C. Since C takes n sorts as parameters, the helper function $[\![C]\!]$ takes n arguments.

$$\texttt{symbol}\ [\![C]\!]\texttt{(}\underbrace{\texttt{\#Sort\{\}}, \ldots, \texttt{\#Sort\{\}}}_{n \text{ times}}\texttt{):\#Sort\{\}}\ \texttt{[]}$$

The second declaration is an axiom declaration that says the helper function $[\![C]\!]$ is indeed just a helper function

$$\texttt{axiom\{\#}s\texttt{\}}\ \texttt{\textbackslash equals\{\#Sort\{\},\#}s\texttt{\}(}[\![C\{s_1, \ldots, s_n\}]\!]\texttt{,}[\![C\{s_1, \ldots, s_n\}]\!]_{verbose}\texttt{)}\ \texttt{[]}$$

where \#s is a fresh meta-level sort variable.

The third declaration is an axiom declaration that says the parametric sort $C\{s_1, \ldots, s_n\}$ is declared iff all sort variables $v_1, \ldots, v_m$ are declared.

$$\begin{array}{l} \texttt{axiom\{\#}s\texttt{\}} \\ \quad \texttt{\textbackslash implies\{\#s\}(} \\ \quad\quad \texttt{\#sortsDeclared\{\#s\}(}[\![s_1, \ldots, s_n]\!]_{list}\texttt{),} \\ \quad\quad\ \texttt{\#sortDeclared\{\#s\}(}[\![C\{s_1, \ldots, s_n\}]\!]\texttt{))}\ \texttt{[]} \end{array}$$

where \#s is a fresh meta-level sort variable.

9.2.5 Lift Object Head Constructors to Meta Symbols

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

$$\begin{aligned}&[\![\_]\!]\colon \langle \textit{object-head-constructor}\rangle \to \langle \textit{meta-head}\rangle\\ &[\![C]\!] = \text{`}\texttt{\#}\text{'}\ \text{`}\texttt{(}\text{'}\ C\ \text{`}\texttt{\{\}}\text{'} \quad \text{for any object head constructor } C\end{aligned}$$

9.2.6 Lift Object Symbol Declarations

An object symbol declaration is lifted to a symbol declaration and two axiom declarations.

$$[\![\_]\!]\colon \langle \textit{object-symbol-declaration}\rangle \to \langle \textit{declaration}\rangle^3$$

Consider the symbol declaration

$$\texttt{symbol}\ C\{v_1,\ldots,v_m\}(s_1,\ldots,s_n)\texttt{:}s\ \texttt{[}\textit{att}\texttt{]}$$

where C is an object symbol constructor, $v_1,\ldots,v_m$ are sort variables, $s_1,\ldots,s_n$ and s are argument sorts and return sort which possibly use sort variables $v_1,\ldots,v_m$.

The first declaration is a symbol declaration for the corresponding helper function $[\![C]\!]$ of the symbol constructor C. Since C takes m sort parameters and n arguments, the helper function $[\![C]\!]$ takes in total $m+n$ arguments.

$$\texttt{symbol}\ [\![C]\!]\texttt{(}\underbrace{\texttt{\#Sort\{\}},\ldots,\texttt{\#Sort\{\}}}_{m\text{ times}}\texttt{,}\underbrace{\texttt{\#Pattern\{\}},\ldots,\texttt{\#Pattern\{\}}}_{n\text{ times}}\texttt{):\#Pattern\{\}}\ \texttt{[]}$$

The second declaration is an axiom declaration that says the helper function $[\![C]\!]$ is indeed just a helper function. Before we show the axiom declaration, let us define some notations. Let us use σ to denote the following meta-pattern

$$\sigma = \texttt{\#symbol\{\}(}[\![C]\!]_{\textit{name}}\texttt{,}[\![v_1,\ldots,v_m]\!]_{\textit{list}}\texttt{,}[\![s_1,\ldots,s_n]\!]_{\textit{list}}\texttt{,}[\![s]\!]\texttt{)}$$

We also need n fresh and distinct meta-level variables of sort `#Pattern`, denoted as $\varphi_1,\ldots,\varphi_n$ respectively. For example when $n=3$, we can choose $\varphi_1,\varphi_2,\varphi_3$ to be

$$\begin{aligned}\varphi_1 &= \texttt{\#P1:\#Pattern\{\}}\\ \varphi_2 &= \texttt{\#P2:\#Pattern\{\}}\\ \varphi_3 &= \texttt{\#P3:\#Pattern\{\}}\end{aligned}$$

The last notation we need to introduce is $[\![\varphi_1,\ldots,\varphi_n]\!]_{\textit{list}}$, which is a meta-pattern of sort `#PatternList` obtained from a list of meta-patterns of sort `#Pattern`. It is defined inductively as follows

$$\begin{aligned}&[\![\_]\!]_{\textit{list}}\colon \langle \textit{meta-pattern}\rangle^* \to \langle \textit{meta-pattern}\rangle\\ &[\![\alpha]\!]_{\textit{list}} = \begin{cases}\text{`}\texttt{\#nilPatternList\{\}}\text{'} & \text{if } \alpha \text{ is empty}\\ \text{`}\texttt{\#consPatternList\{\}}\text{'}\ \text{`}\texttt{(}\text{'}\ \varphi\ \text{`}\texttt{,}\text{'}\ [\![\beta]\!]_{\textit{list}}\ \text{`}\texttt{)}\text{'} & \text{if } \alpha = \varphi,\beta\end{cases}\end{aligned}$$

Now we are ready to show the axiom declaration which says the helper function $[\![C]\!]$ is indeed just a helper function.

$$\texttt{axiom\{\#}s\texttt{\}}$$
$$\quad\texttt{\textbackslash equals\{\#Pattern\{\},\#}s\texttt{\}(}$$
$$\quad\quad[\![C]\!]([\![v_1]\!],\ldots,[\![v_m]\!],\varphi_1,\ldots,\varphi_n)\texttt{,}$$
$$\quad\quad\texttt{\#application\{\}(}\sigma,[\![\varphi_1,\ldots,\varphi_n]\!]_{list}\texttt{) []}$$

where #s is a fresh meta-level sort variable.

The third declaration is an axiom declaration that says the parametric symbol $C\{v_1,\ldots,v_m\}$ is declared iff all sort variables $v_1,\ldots,v_m$ are declared.

$$\texttt{axiom\{\#}s\texttt{\}}$$
$$\quad\texttt{\textbackslash implies\{\#}s\texttt{\}(}$$
$$\quad\quad\texttt{\#sortsDeclared\{\#}s\texttt{\}(}[\![v_1,\ldots,v_m]\!]_{list}\texttt{),}$$
$$\quad\quad\texttt{\#symbolDeclared\{\{\}\#}s\texttt{\}(}\sigma\texttt{)) []}$$

where #s is a fresh meta-level sort variable.

9.2.7 Lift Object Alias Declarations

An object alias declaration is lifted to a symbol declaration.

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

Consider the alias declaration

$$\texttt{alias}\ C\{v_1,\ldots,v_m\}(s_1,\ldots,s_n)\texttt{:}s\ \texttt{[}\textit{att}\texttt{]}$$

where C is an object alias constructor, $v_1,\ldots,v_m$ are sort variables, $s_1,\ldots,s_n$ and s are argument sorts and return sort which possibly use sort variables $v_1,\ldots,v_m$.

The alias declaration is lifted to a symbol declaration for the corresponding helper function $[\![C]\!]$ of the alias constructor C. Since C takes m sort parameters and n arguments, the helper function $[\![C]\!]$ takes in total $m+n$ arguments.

$$\texttt{symbol}\ [\![C]\!]\texttt{(}\underbrace{\texttt{\#Sort\{\},}\ldots\texttt{,\#Sort\{\}}}_{m\text{ times}}\texttt{,}\underbrace{\texttt{\#Pattern\{\},}\ldots\texttt{,\#Pattern\{\}}}_{n\text{ times}}\texttt{):\#Pattern\{\} []}$$

9.2.8 Lift Patterns

We remind readers of the grammar of Kore patterns in Section 9.1.4. In this section, we define the lifting process for patterns

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

Variables. We first define an auxiliary lifting process that lifts an object variable to a meta-pattern of sort `#Variable`.

$$[\![\_]\!]_{var}\colon\langle\textit{object-variable}\rangle\to\langle\textit{meta-pattern}\rangle$$
$$[\![x\texttt{:}s]\!]=\texttt{\#variable\{\}(}[\![x]\!]_{name}\texttt{,}[\![s]\!]\texttt{)}\quad\text{for any object variable } x\texttt{:}s$$

The lifting of a variable v is defined as

$$[\![v]\!]=\begin{cases}\texttt{\#variableAsPattern\{\}(}[\![v]\!]_{var}\texttt{)} & \text{if } v \text{ is } \langle\textit{object-variable}\rangle\\ v & \text{if } v \text{ is } \langle\textit{meta-variable}\rangle\end{cases}$$

Patterns Constructed with Heads. The lifting of a pattern of the form $C\{s_1, \ldots, s_m\}(\varphi_1, \ldots, \varphi_n)$ is defined as

$$[\![C\{s_1, \ldots, s_n\}(\varphi_1, \ldots, \varphi_n)]\!] = \begin{cases} [\![C]\!]([\![s_1]\!], \ldots, [\![s_n]\!], [\![\varphi_1]\!], \ldots, [\![\varphi_n]\!]) & \text{if } C \text{ is } \langle \textit{object-head} \rangle \\ C\{s_1, \ldots, s_n\}([\![\varphi_1]\!], \ldots, [\![\varphi_n]\!]) & \text{if } C \text{ is } \langle \textit{meta-head} \rangle \end{cases}$$

Patterns Constructed with Logic Connectives. The lifting of a pattern constructed using logic connectives is defined as follows.

$[\![\texttt{\textbackslash and}\{s\}(\varphi_1,\varphi_2)]\!] = \texttt{\#\textbackslash and}\{\}([\![s]\!],[\![\varphi_1]\!],[\![\varphi_2]\!])$ if s is $\langle \textit{object-sort} \rangle$
$[\![\texttt{\textbackslash and}\{s\}(\varphi_1,\varphi_2)]\!] = \texttt{\textbackslash and}\{s\}([\![\varphi_1]\!],[\![\varphi_2]\!])$ if s is $\langle \textit{meta-sort} \rangle$
$[\![\texttt{\textbackslash not}\{s\}(\varphi)]\!] = \texttt{\#\textbackslash not}\{\}([\![s]\!],[\![\varphi]\!])$ if s is $\langle \textit{object-sort} \rangle$
$[\![\texttt{\textbackslash not}\{s\}(\varphi_1)]\!] = \texttt{\textbackslash not}\{s\}([\![\varphi]\!])$ if s is $\langle \textit{meta-sort} \rangle$
$[\![\texttt{\textbackslash or}\{s\}(\varphi_1,\varphi_2)]\!] = \texttt{\#\textbackslash or}\{\}([\![s]\!],[\![\varphi_1]\!],[\![\varphi_2]\!])$ if s is $\langle \textit{object-sort} \rangle$
$[\![\texttt{\textbackslash or}\{s\}(\varphi_1,\varphi_2)]\!] = \texttt{\textbackslash or}\{s\}([\![\varphi_1]\!],[\![\varphi_2]\!])$ if s is $\langle \textit{meta-sort} \rangle$
$[\![\texttt{\textbackslash implies}\{s\}(\varphi_1,\varphi_2)]\!] = \texttt{\#\textbackslash implies}\{\}([\![s]\!],[\![\varphi_1]\!],[\![\varphi_2]\!])$ if s is $\langle \textit{object-sort} \rangle$
$[\![\texttt{\textbackslash implies}\{s\}(\varphi_1,\varphi_2)]\!] = \texttt{\textbackslash implies}\{s\}([\![\varphi_1]\!],[\![\varphi_2]\!])$ if s is $\langle \textit{meta-sort} \rangle$
$[\![\texttt{\textbackslash iff}\{s\}(\varphi_1,\varphi_2)]\!] = \texttt{\#\textbackslash iff}\{\}([\![s]\!],[\![\varphi_1]\!],[\![\varphi_2]\!])$ if s is $\langle \textit{object-sort} \rangle$
$[\![\texttt{\textbackslash iff}\{s\}(\varphi_1,\varphi_2)]\!] = \texttt{\textbackslash iff}\{s\}([\![\varphi_1]\!],[\![\varphi_2]\!])$ if s is $\langle \textit{meta-sort} \rangle$
$[\![\texttt{\textbackslash forall}\{s\}(v,\varphi)]\!] = \texttt{\#\textbackslash forall}\{\}([\![s]\!],[\![v]\!]_{var},[\![\varphi]\!])$ if s is $\langle \textit{object-sort} \rangle$
$[\![\texttt{\textbackslash forall}\{s\}(v,\varphi)]\!] = \texttt{\textbackslash forall}\{s\}([\![v]\!]_{var},[\![\varphi]\!])$ if s is $\langle \textit{meta-sort} \rangle$
$[\![\texttt{\textbackslash exists}\{s\}(v,\varphi)]\!] = \texttt{\#\textbackslash exists}\{\}([\![s]\!],[\![v]\!]_{var},[\![\varphi]\!])$ if s is $\langle \textit{object-sort} \rangle$
$[\![\texttt{\textbackslash exists}\{s\}(v,\varphi)]\!] = \texttt{\textbackslash exists}\{s\}([\![v]\!]_{var},[\![\varphi]\!])$ if s is $\langle \textit{meta-sort} \rangle$
$[\![\texttt{\textbackslash ceil}\{s_1,s_2\}(\varphi)]\!] = \texttt{\#\textbackslash ceil}\{\}([\![s_1]\!],[\![s_2]\!],[\![\varphi]\!])$ if s_1, s_2 are $\langle \textit{object-sort} \rangle$
$[\![\texttt{\textbackslash ceil}\{s_1,s_2\}(\varphi)]\!] = \texttt{\textbackslash ceil}\{s_1,s_2\}([\![\varphi]\!])$ if s_1, s_2 are $\langle \textit{meta-sort} \rangle$
$[\![\texttt{\textbackslash floor}\{s_1,s_2\}(\varphi)]\!] = \texttt{\#\textbackslash floor}\{\}([\![s_1]\!],[\![s_2]\!],[\![\varphi]\!])$ if s_1, s_2 are $\langle \textit{object-sort} \rangle$
$[\![\texttt{\textbackslash floor}\{s_1,s_2\}(\varphi)]\!] = \texttt{\textbackslash floor}\{s_1,s_2\}([\![\varphi]\!])$ if s_1, s_2 are $\langle \textit{meta-sort} \rangle$
$[\![\texttt{\textbackslash equals}\{s_1,s_2\}(\varphi_1,\varphi_2)]\!] = \texttt{\#\textbackslash equals}\{\}([\![s_1]\!],[\![s_2]\!],[\![\varphi_1]\!],[\![\varphi_2]\!])$ if s_1, s_2 are $\langle \textit{object-sort} \rangle$
$[\![\texttt{\textbackslash equals}\{s_1,s_2\}(\varphi_1,\varphi_2)]\!] = \texttt{\textbackslash equals}\{s_1,s_2\}([\![\varphi_1]\!],[\![\varphi_2]\!])$ if s_1, s_2 are $\langle \textit{meta-sort} \rangle$
$[\![\texttt{\textbackslash in}\{s_1,s_2\}(\varphi_1,\varphi_2)]\!] = \texttt{\#\textbackslash in}\{\}([\![s_1]\!],[\![s_2]\!],[\![\varphi_1]\!],[\![\varphi_2]\!])$ if s_1, s_2 are $\langle \textit{object-sort} \rangle$
$[\![\texttt{\textbackslash in}\{s_1,s_2\}(\varphi_1,\varphi_2)]\!] = \texttt{\textbackslash in}\{s_1,s_2\}([\![\varphi_1]\!],[\![\varphi_2]\!])$ if s_1, s_2 are $\langle \textit{meta-sort} \rangle$
$[\![\texttt{\textbackslash top}\{s\}()]\!] = \texttt{\#\textbackslash top}\{\}([\![s]\!])$ if s is $\langle \textit{object-sort} \rangle$
$[\![\texttt{\textbackslash top}\{s\}()]\!] = \texttt{\textbackslash top}\{s\}()$ if s is $\langle \textit{meta-sort} \rangle$
$[\![\texttt{\textbackslash bottom}\{s\}()]\!] = \texttt{\#\textbackslash bottom}\{\}([\![s]\!])$ if s is $\langle \textit{object-sort} \rangle$
$[\![\texttt{\textbackslash bottom}\{s\}()]\!] = \texttt{\textbackslash bottom}\{s\}()$ if s is $\langle \textit{meta-sort} \rangle$

9.2.9 Lift Axiom Declarations

In Kore, an axiom declaration is lifted as follows.

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

$$[\![\texttt{axiom}\{s_1,\ldots,s_n\}\ \ \varphi]\!] = \begin{cases} \texttt{axiom}\{s'_1,\ldots,s'_m\}\ \texttt{\textbackslash implies}\{s'\}(\\ \qquad \texttt{\#sortDeclared}\{s'\}([\![s''_1,\ldots,s''_k]\!]_{list}),\ \texttt{\#provable}\{s'\}([\![\varphi]\!])) \\ \qquad\qquad \text{if } \varphi \text{ is } \langle \textit{object-pattern}\rangle \\ \texttt{axiom}\{s'_1,\ldots,s'_m\}\ \texttt{\textbackslash implies}\{s'\}(\\ \qquad \texttt{\#sortDeclared}\{s'\}([\![s''_1,\ldots,s''_k]\!]_{list}), [\![\varphi]\!]) \\ \qquad\qquad \text{if } \varphi \text{ is } \langle \textit{meta-pattern}\rangle \end{cases}$$

where $s'_1,\ldots,s'_m$ are all $\langle \textit{meta-sort-variable}\rangle$ in $s_1,\ldots,s_n$, $s''_1,\ldots,s''_k$ are all $\langle \textit{object-sort-variable}\rangle$ in $s_1,\ldots,s_n$, and s' is a $\langle \textit{meta-sort-variable}\rangle$. In the case when φ is a $\langle \textit{object-pattern}\rangle$, s' is a fresh $\langle \textit{meta-sort-variable}\rangle$ that is different from $s'_1,\ldots,s'_m$. In the case when φ is a $\langle \textit{meta-pattern}\rangle$, s' is the sort of the $\langle \textit{meta-pattern}\rangle$ $[\![\varphi]\!]$. Depending on the the concrete form of $[\![\varphi]\!]$, s' may be one of the parameters $s'_1,\ldots,s'_m$, or the one built using one of the $\langle \textit{meta-sort-constructor}\rangle$ defined in Section 9.1.2.

10 Conclusion and Future Work

It took us a long journey to realize that $\mathbb{K}$ is a fragment of matching logic and all $\mathbb{K}$ does is logic reasoning with some matching logic theories

$$\underbrace{T_{\text{binders}} \cup T_{\text{fixed-points}} \cup T_{\text{contexts}} \cup T_{\text{rewriting}} \cup \cdots \cup T_{\text{user-defined}} \vdash \cdots}_{\mathbb{K} \text{ is a best effort realization of matching logic reasoning}}$$

In Section 3 and 4, we showed a few important case studies of these theories and pointed out the fact that it is often not trivial to specify those theories because they are infinite artifacts. In Section 6 we proposed an approach that provides us a one-stop solution of specifying recursively enumerable theories in a nice and mathematically crystal clear manner, by developing a deep embedding of matching logic in a finite fragment of itself, which we call the meta-theory K. Section 7 and 8 define the meta-theory K and explain how to encode theories in it. Section 9 defines the Kore language including both its syntax and semantics.

Future Work. This document does not answer all questions, and there is still much to be grained and much future research work to do. We think the future work can be divided into three main categories.

The first category of future work is to define $\mathbb{K}$ in the Kore language. In Section 4, we defined a few important underlying theories of $\mathbb{K}$ using natural language and mathematical notations. In there, our main purpose is to demonstrate that $\mathbb{K}$ is all about matching logic reasoning. In the future, all features and functionalities of $\mathbb{K}$ will be specified as Kore definitions. These include everything that we mentioned in Section 4, and also all $\mathbb{K}$'s APIs and commands such as `kompile`, `krun`, and `kprove`. These Kore definitions plus this document forms a solid answer to the ultimate question: "what is the semantics of $\mathbb{K}$?".

The second category of future work is to extend the Kore language and to develop tools for it. The Kore language has a rather simple syntax. During its design, we considered more on machine readability and relative less on human readability. As a result, writing Kore definitions manually is definitely possible, but it can be tedious and error proneing. As we define $\mathbb{K}$ in Kore, we will gradually notice that certain syntactic sugar should be introduced to Kore so that the resulting Kore definitions are shorter. For example, empty parenthesis and braces in Kore patterns and Kore declarations need not be written explicitly all the time. New logic connectives such that `\next` and `\rewrites` can be introduced to Kore as they appear very often in the theory of rewriting and reachability. The general principle of adding new syntax features to Kore is that any syntactic sugar that is added to Kore should be proved to be useful and highly needed in practice, in particular, in the process of defining $\mathbb{K}$ in Kore. In addition, we may need to change the meta-theory in correspondence to our changes to Kore's syntax, in order to keep the lifting process, the process that gives Kore semantics, clean and simple. In the meantime, automatic tools about the Kore language should be developed. These include parsers and pretty-printers, implementations of the lifting process, wellformedness checkers and so on.

The third category of future work is to develop provers for matching logic. More specifically, to develop tools for reasoning with the meta-theory and Kore. These tools include at least the following three kinds. The first is automatic proof checkers. A proof checker takes a Kore definition and a proof obligation, which is often automatically generated by $\mathbb{K}$ back-ends, and a proof for that proof obligation, and checks whether the proof is valid. Proof checkers should be very simple so that their correctness is self-convincing, and very fast so that they can handle proofs of monster sizes. The second is interactive provers. An interactive prover takes a Kore definition and a proof obligation, and provides an interactive proving environment for human users to input proof hints and tactics, and finds a proof for the proof obligation that can be checked by proof checkers. Interactive provers aim at completeness, which means they should be able to prove any valid claims, at least in theory. The third is automatic provers. An automatic prover automatically finds a proof for a constrained type of Kore definitions and proof obligations. Automatic provers often care less about completeness but more about performance. Apart from developing the above three kinds of provers, we also need to consider how to generate proof obligations and how to represent proofs in a structured and well-organized manner.

11 Proof of Faithfulness Theorem

Warning. The proof here is outdated. However, the general approach is still valid. Read with cautions.

We restate the faithfulness theorem

Theorem 12 (Faithfulness Theorem)**.** *Given a matching logic theory T and a naming function e. Let $[\![T]\!]$ be the meta-theory instantiated with T, and φ be any pattern in T,*

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

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 10, 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:

$$\begin{array}{l} [\![T]\!] \text{ is the matching logic theory } (\Sigma, [\![A]\!]) \\ [\![A]\!] = \{[\![\psi]\!] \mid \psi \in A\} \\ [\![\varphi]\!] = \begin{cases} x:s & \text{if } \varphi \text{ is } \texttt{\#variable}(x, s) \\ \sigma([\![\varphi_1]\!], \ldots, [\![\varphi_n]\!]) & \text{if } \varphi \text{ is } \texttt{\#application}(\sigma, (\varphi_1, \ldots, \varphi_n)) \\ [\![\varphi_1]\!] \wedge [\![\varphi_2]\!] & \text{if } \varphi \text{ is } \texttt{\#\textbackslash and}(\varphi_1, \varphi_2, s), \\ \neg [\![\varphi_1]\!] & \text{if } \varphi \text{ is } \texttt{\#\textbackslash not}(\varphi_1, s) \\ \exists x.[\![\varphi_1]\!] & \text{if } \varphi \text{ is } \texttt{\#\textbackslash exists}(x, s_1, \varphi_1, s_2) \end{cases} \end{array}$$

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 10.

Proof Sketch of Theorem 10.

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] J. Bergstra and J. Tucker. Equational specifications, complete term rewriting systems, and computable and semicomputable algebras. *Journal of the Association for Computing Machinery*, 42(6):1194–1230, 1995.

[2] Y. Bertot and P. Castéran. *Interactive theorem proving and program development: CoqArt: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

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

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

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

[10] J. A. Goguen. Topology and category theory in computer science. chapter Types As Theories, pages 357–385. Oxford University Press, Inc., New York, NY, USA, 1991.

[11] J. A. Goguen and J. Meseguer. Order-sorted algebra i: equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2):217 – 273, 1992.

[12] 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`.

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

[14] 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.

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

[16] L. Kovács and A. Voronkov. *First-Order Theorem Proving and Vampire*, pages 1–35. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

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

[18] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

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

[20] L. C. Paulson. *Isabelle: A generic theorem prover*, volume 828. Springer Science & Business Media, 1994.

[21] 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.

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

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

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

[25] 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.

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