Introduction to Matching Logic

Runtime Verification

November 12, 2020

Contents

1 Introduction

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

Matching logic allows us to define *theories* (S, Σ, A) consisting of potentially infinite sets of *sorts* S, of *symbols* Σ over sorts in S (also called S-symbols), and of *patterns* A built with symbols in Σ (also called

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

... ...

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

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 Matching Logic

In this section we first recall basic matching logic syntax and semantics notions from [16] at a theoretical level.

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 $\textsc{Pattern}_s$ of patterns of sort $s \in S$ is generated by the following grammar:

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

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 $\textsc{Pattern}$ be the S-sorted set of patterns $\{\textsc{Pattern}_s\}_{s \in S}$. 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 $Mod(S, \Sigma)$, or $Mod(\Sigma)$ when S is understood, denote the (category of) models of a signature (S, Σ). Given a model M and a map $\rho : Var \to M$, called an M*-valuation*, let its extension $\overline{\rho} : \textsc{Pattern} \to \mathcal{P}(M)$ be inductively defined as follows:

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

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

Pattern φ_s is an M*-predicate*, or a *predicate in* M, iff for any M-valuation $\rho : Var \to M$, it is the case that $\overline{\rho}(\varphi_s)$ is either M_s (it holds) or $\emptyset$ (it does not hold). Pattern φ_s is a *predicate* iff it is a predicate in all models M. For example, $\top_s$ and $\bot_s$ are predicates, and if φ, φ_1 and φ_2 are predicates then so are $\neg\varphi$, $\varphi_1 \wedge \varphi_2$, and $\exists x\,.\,\varphi$. That is, the logical connectives of matching logic preserve the predicate nature of patterns. 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 $\rho : Var \to M$. 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 $Mod(T)$, or $Mod(S, \Sigma, A)$ or $Mod(\Sigma, A)$, also denoted by $[\![T]\!]$, or $[\![(S, \Sigma, A)]\!]$ or $[\![(\Sigma, A)]\!]$, be its (category of) models $\{M \mid M \in Mod_\Sigma,\ M \models_\Sigma A\}$.

A signature (S', Σ') is called a *subsignature* of (S, Σ), written $(S', \Sigma') \hookrightarrow (S, \Sigma)$, if and only if $S' \subseteq S$ and $\Sigma' \subseteq \Sigma$. If $M \in Mod(\Sigma)$ then we let $M \restriction_{\Sigma'} \in Mod(\Sigma')$ denote its Σ'*-reduct*, or simply its *reduct* when Σ' is understood, defined as follows: $(M \restriction_{\Sigma'})_{s'} = M_{s'}$ for any $s' \in S'$ and $\sigma'_{M\restriction_{\Sigma'}} = \sigma'_M$ for any $\sigma' \in \Sigma'$. It

may help to think of signatures as interfaces and of models as *implementations* of such interfaces. Indeed, models provide concrete values for each sort, and concrete relations for symbols. Then the reduct $M{\restriction_{\Sigma'}}$ can be regarded as a "wrapper" of the implementation M of Σ turning it into an implementation of Σ', or a reuse of a richer implementation in a smaller context.

2.3 Useful Symbols and Notations

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

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

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

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 $\varphi \in \textsc{Pattern}_{s_1}$ then $\lceil \varphi \rceil^{s_2}_{s_1}$ is a predicate which holds iff φ is defined: if $\rho : \mathit{Var} \to M$ then $\overline{\rho}(\lceil \varphi \rceil^{s_2}_{s_1})$ is either $\emptyset$ (i.e., $\overline{\rho}(\bot_{s_2})$) when $\overline{\rho}(\varphi) = \emptyset$ (i.e., φ undefined in ρ), or is M_{s_2} (i.e., $\overline{\rho}(\top_{s_2})$) when $\overline{\rho}(\varphi) \neq \emptyset$ (i.e., φ defined).

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

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

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

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

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

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

Similarly, we can define *membership*: if $x \in \mathit{Var}_{s_1}$, $\varphi \in \textsc{Pattern}_{s_1}$ and $s_1, s_2 \in S$, 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 $\rho : \mathit{Var} \to M$, $\overline{\rho}(x \in^{s_2}_{s_1} \varphi) = \emptyset$ iff $\rho(x) \notin \overline{\rho}(\varphi)$, and $\overline{\rho}(x \in^{s_2}_{s_1} \varphi) = M_{s_2}$ iff $\rho(x) \in \overline{\rho}(\varphi)$. It is convenient to extend the definition of membership to cases when the first component is not a variable, but a pattern: if $\psi, \varphi \in \textsc{Pattern}_{s_1}$, 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 $\lceil \_ \rceil^{s_2}_{s_1}$ or $\lfloor \_ \rfloor^{s_2}_{s_1}$, respectively, and similarly for the equality and membership. Also, if the sort decorations cannot be inferred from context, then we assume the stated property/axiom/rule holds for all such sorts. For example, the generic pattern axiom "$\lceil x \rceil$ where $x \in \mathit{Var}$" replaces all the axioms $\lceil x : s_1 \rceil^{s_2}_{s_1}$ above for all the definedness symbols for all the sorts s_1 and s_2.

Refer to this later when we talk about parameters.

Note that, by default, symbols are interpreted as relations in matching logic. It is often the case, though, that we want symbols to be interpreted as *functions*. This can be easily done by axiomatically constraining those symbols to evaluate to singletons. For example, if f is a unary symbol, then the pattern equation "$\forall x \,.\, \exists y \,.\, f(x) = y$" (the convention for free variables allows us to drop the universal quantifier) states that in any model M, the set $f_M(a)$ contains precisely one element for any $a \in M$. Inspired from similar notations in other logics, we adopt the familiar notation "$\sigma : s_1 \times \cdots \times s_n \rightarrow s$" 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 [16].

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 [16].

2.4 Sound and Complete Deduction

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. $\varphi_1 \rightarrow (\varphi_2 \rightarrow \varphi_1)$ (PROPOSITIONAL$_1$)

2. $(\varphi_1 \rightarrow (\varphi_2 \rightarrow \varphi_3)) \rightarrow ((\varphi_1 \rightarrow \varphi_2) \rightarrow (\varphi_1 \rightarrow \varphi_3))$ (PROPOSITIONAL$_2$)

3. $(\neg\varphi_1 \rightarrow \neg\varphi_2) \rightarrow (\varphi_2 \rightarrow \varphi_1)$ (PROPOSITIONAL$_3$)

4. $\dfrac{\varphi_1 \quad \varphi_1 \rightarrow \varphi_2}{\varphi_2}$ (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 some of the proof rules that deal with quantifiers from first order logic. Notice that as explained in [16], the FOL substitution proof rule does not hold in its general form in matching logic. Instead, we only have *variable substitution* in matching logic.

First-order logic proof rules:

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

6. $\dfrac{\varphi}{\forall x\,.\,\varphi}$ (UNIVERSAL GENERALIZATION)

7. $\vdash (\forall x\,.\,\varphi) \rightarrow \varphi[y/x]$ (VARIABLE SUBSTITUTION)

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

Propagation rules:

8. $\vdash \sigma(\varphi_1, \ldots, \varphi_{i-1}, \bot, \varphi_{i+1}, \ldots, \varphi_n) \rightarrow \bot$ (PROPAGATION$_\bot$)

9. $\sigma(\varphi_1, \ldots, \varphi_{i-1}, \varphi_i \vee \varphi_i', \varphi_{i+1}, \ldots, \varphi_n) \rightarrow \sigma(\varphi_1, \ldots, \varphi_{i-1}, \varphi_i, \varphi_{i+1}, \ldots, \varphi_n) \vee \sigma(\varphi_1, \ldots, \varphi_{i-1}, \varphi_i', \varphi_{i+1}, \ldots, \varphi_n)$ (PROPAGATION$_\vee$)

10. $\sigma(\varphi_1, \ldots, \varphi_{i-1}, \exists x.\varphi_i, \varphi_{i+1}, \ldots, \varphi_n) \rightarrow \exists x.\sigma(\varphi_1, \ldots, \varphi_{i-1}, \varphi_i, \varphi_{i+1}, \ldots, \varphi_n)$ when $x \notin \bigcup_{j \neq i} FV(\varphi_j)$ (PROPAGATION$_\exists$)

Frame rule:

11. $\dfrac{\varphi_i \rightarrow \varphi_i'}{\sigma(\varphi_1, \ldots, \varphi_{i-1}, \varphi_i, \varphi_{i+1}, \ldots, \varphi_n) \rightarrow \sigma(\varphi_1, \ldots, \varphi_{i-1}, \varphi_i', \varphi_{i+1}, \ldots, \varphi_n)}$ (FRAMING)

Besides the above rules, we need two more proof rules to make the proof system a complete one. These two rules are

More rules:

12. $\vdash \exists x.x$ (Existence)

To ease our notation in writing the next proof rule, let us introduce *symbol context*. We abbreviate $\sigma(\varphi_1, \ldots, \varphi_{i-1}, \varphi_i, \varphi_{i+1}, \ldots, \varphi_n)$ as $C_{\sigma,i}[\varphi_i]$. When the position i is not important, we omit it and write $C_\sigma[\varphi_i]$. We inductively define what a *symbol context* is. A symbol context C is either the identity context, i.e., $C[\varphi] = \varphi$, or $C[\varphi] = C_\sigma[C'[\varphi]]$ where C' is a symbol context. In other words, $C[\varphi]$ is a symbol context if the path of the occurrence of φ in $C[\varphi]$ contains only symbols and no logic connectives.

Using our notation of symbol contexts, we present the last proof rule which captures the fact that variables are singletons:

13. $\vdash \neg(C_1[x \wedge \varphi] \wedge C_2[x \wedge \neg\varphi])$ where C_1, C_2 are symbol contexts (Singleton Variables)

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

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

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

3 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.

3.1 Binders

More recent research shows that binders can be defined as functional symbols. Change section 4.1–4.3 to reflect that.

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 **??**, 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 on the other hand it builds a term. Fortunately, matching logic allows us to separate these two operations, and use the builtin existential

quantifier for binding. Specifically, we define a symbol λ^0 and regard λ as syntactic sugar for the pattern that existentially binds its first argument into λ^0:

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

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

Explain why λ is not a symbol, but an alias

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

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

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

essentially for free, because matching logic's builtin existential quantifier and substitution already enjoy the α-equivalence property [16]. The β-equality, on the other hand, requires an important side condition. To start the discussion, let us suppose that we naively define it as follows:

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

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

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

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

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

3.2 Fixed Points

Similarly to the λ-binder in Section 3.1, 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 3.1 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 $\bar{F}(A) = M \setminus (F(M \setminus A))$.

3.3 Contexts

Matching logic allows us to define a very general notion of context. Our contexts can be used not only to define evaluation strategies of various language constructs, like how evaluation contexts are traditionally used [6], but also for configuration abstraction to enhance modularity and reuse, and for matching multiple sub-patterns of interest at the same time.

Brandon: how about the locality principle?

Like λ in Section 3.1, 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:

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

And the following axiom schema for composing nested contexts:

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

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

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

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

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

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

3.3.1 Evaluation Strategies of Language Constructs

Suppose that a programming language has an if-then-else statement, say $\mathsf{ite} \in \Sigma_{BExp \times Stmt \times Stmt, Stmt}$, whose evaluation strategy is to first evaluate its first argument and then, depending on whether it evaluates to *true* or *false*, to rewrite to either its second argument or its third argument. We here only focus on its evaluation strategy and not its reduction rules; the latter will be discussed in Section 3.4. Assuming that all reductions/rewrites apply in context, as discussed in Section 3.4, we can state that ite is given permission to apply reductions within its first argument with the following axiom:

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

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

As an example, suppose that besides ite with strategy strict(1) we also have an infix operation $\_ < \_ \in \Sigma_{AExp \times AExp, BExp}$ 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:

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

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

$$\gamma\, \square_1\, .((\gamma\, \square_2\, .\mathsf{ite}(\square_2, S_1, S_2))[(\gamma\, \square_3\, .1 < \square_3)[\square_1]])[x]$$

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

$$\begin{array}{rcll}
 & & \gamma\, \square_1\, .((\gamma\, \square_2\, .\mathsf{ite}(\square_2, S_1, S_2))[(\gamma\, \square_3\, .1 < \square_3)[\square_1]])[x] & \\
 & = & \gamma\, \square_1\, .((\gamma\, \square_2\, .\mathsf{ite}(\square_2, S_1, S_2))[1 < \square_1])[x] & /\!/\ \text{strict(2) axiom for } _ < _ \\
 & = & \gamma\, \square_1\, .(\mathsf{ite}(1 < \square_1, S_1, S_2))[x] & /\!/\ \text{strict(1) axiom for ite}
\end{array}$$

Therefore, $\mathsf{ite}(1 < x, S_1, S_2)$ can be matched against a pattern of the form $C[x]$, where C, as expected, is $\gamma\, \square_1\, .(\mathsf{ite}(1 < \square_1, S_1, S_2))$, a context of sort $Context\{AExp, Stmt\}$. That is, x has been "pulled" out of the ite context. We point out that the above is a typical example of reasoning about contexts in matching logic. One applies a sequence of strictness axioms, from left to right, to pull out the redex. Then, the resulting nested contexts can be combined and merged to one uniform context with the axiom for nested context. Finally, the body of the uniform context can be simplified by applying the sequence of strictness axiom from right to left.

Now other semantic rules or axioms can be applied to reduce x, by simply matching x in a context. At any moment during the reduction, the axioms above can be applied backwards and thus whatever x reduces to can be "plugged" back into its context. This way, the axiomatic approach to contexts in matching logic achieves the "pull and plug" mechanism underlying reduction semantics with evaluation contexts [6] by means of logical deduction using the generic sound and complete proof system in Section 2.4. Also, notice that our notion of context is more general than that in reduction semantics. That is, it is not only used for reduction or in order to isolate a redex to be reduced, but it can be used for matching any relevant data from a program configuration. More examples below will illustrate that.

3.3.2 Multi-Hole Contexts and Configuration Abstraction

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

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

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

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

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

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

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:

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

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

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

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

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

Here, the first, the second, and the fourth equations are by context reasoning as in Section 3.3.1 while the third and the fifth equations are by multi-hole context definition. 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.

3.4 Rewriting and Reachability

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

Let us first note that, in matching logic, giving a binary relation $R_s \in M_s \times M_s$ on the carrier of sort s of a model M is equivalent to interpreting a unary symbol $\bullet \in \Sigma_{s,s}$ in M: indeed, for any $a, b \in M_s$, take $a\ R_s\ b$ iff $a \in \bullet_M(b)$. If the intuition for $a\ R_s\ b$ is "state a transitions to state b", then the intuition for $\bullet_M$ is that of "transition to" or "next": $\bullet_M(b)$ 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 $\varphi \rightarrow \bullet\varphi'$. We have $M \models \varphi \rightarrow \bullet\varphi'$ iff $\overline{\rho}(\varphi) \subseteq \bullet_M(\overline{\rho}(\varphi'))$ 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:

$$\begin{aligned}
&\bullet\{s\} \in \Sigma_{s,s} && /\!/ \text{ a "next" symbol schema} \\
&\varphi \Rightarrow\!\{s\}\ \varphi' \equiv \varphi \rightarrow \bullet\{s\}\varphi' && /\!/ \text{ a "rewrite relation" alias schema}
\end{aligned}$$

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 3.3.2, 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 $\bullet$ symbol:

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

Together with structural framing [16], these axioms allow not only to lift multiple local rewrites that are part of the same rule into one rewrite higher in the pattern, but also to combine multiple parallel rewrites into one rewrite, thus giving natural support for *true concurrency*. Note that the axioms above allow to lift the $\bullet$ symbol from any proper subset of orthogonal subpatterns, without enforcing the lifting of *all* the $\bullet$ 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:

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

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

3.5 An Example

Consider the following example. Suppose we want to define a simple transition system whose state/configuration set is $\mathbb{N} \cup \{\mathsf{rand}\}$, where rand is the program construct representing a random number generator. The intended semantics of rand is that it can go to any number $n \in \mathbb{N}$ in one step. How can we write an axiom that captures the intended semantics?

We here give three candidate axioms.

$$\begin{array}{lll} \text{(Axiom A)} & \forall x.(\mathsf{rand} \rightarrow (x > 0 \rightarrow \bullet(x))) & /\!/ \text{ correct} \\ \text{(Axiom B)} & \mathsf{rand} \rightarrow \exists x.(x > 0 \wedge (\bullet x)) & /\!/ \text{ incorrect} \\ \text{(Axiom C)} & \forall x.(\mathsf{rand} \rightarrow (x > 0 \wedge \bullet(x))) & /\!/ \text{ totally wrong} \end{array}$$

We first explain why (Axiom B) and (Axiom C) are wrong. First of all, (Axiom C) is *inconsistent*, meaning that we can prove everything, even "bottom $\bot$", from (Axiom C). To see why, we simply instantiate the "$\forall x$" with $x = 0$, and then we get $\forall x.\mathsf{rand} \rightarrow (0 > 0 \wedge \bullet(x))$. Since $0 > 0$ is false, meaning it equals $\bot$, we get $\mathsf{rand} \rightarrow \bot$.

(Axiom B) is also wrong, but it is *consistent*. What it says is that rand must be the predecessor of *some positive number*. Therefore, the transition system where rand can *only* go to, say, 42, satisfies (Axiom B). This means that (Axiom B) is too weak to capture the intended semantics of rand.

(Axiom A) is the correct axiom. It says that *for all positive numbers n*, rand is a predecessor of n. This captures precisely the intended semantics of rand. Note that (Axiom A) is a *positive axiom*, meaning that it merely says what the transition system *must do*, but not constrains what it *cannot do*. Positive axioms will *not* introduce inconsistency, as the total transition system (i.e., the translation relation is the total relation and all states can go to all other states, including themselves) is always a candidate model.

To conclude this example, we show that if we want to capture *in further* that rand can *only* go to positive numbers and nowhere else, we need the following axiom, which is *not* a positive axiom anymore:

$$\text{(Axiom D)} \qquad \mathsf{rand} \rightarrow \bigcirc(\exists x.(x > 0) \wedge x)$$

where $\bigcirc\varphi \equiv \neg\bullet\neg\varphi$ is the dual of "one-path next" $\bullet$, called "all-path next". The axiom says that *for all next states* of rand, it must be some positive number. (Axiom A) together with (Axiom D) capture the intended semantics of rand most precisely. For the purpose of reasoning about *what the transition system can do*,

e.g., one-path reachability properties, (Axiom A) is good enough, but for reasoning about more general properties, e.g., all-path reachability properties, we must know *what the transition system cannot do*, and (Axiom D) is needed in there.

4 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.

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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