Matching Logic: Syntax and Semantics

Grigore Roșu[1] and Traian Florin Șerbănuță[2]

[1] University of Illinois at Urbana-Champaign, USA
grosu@illinois.edu
[2] University of Bucharest, Romania
traian.serbanuta@unibuc.ro

Abstract. This paper recalls the syntax and semantics of Matching Logic [6], a first-order logic (FOL) variant for specifying and reasoning about structure by means of patterns and pattern matching. Its sentences, the *patterns*, are constructed using *variables*, *symbols*, *connectives* and *quantifiers*, but no difference is made between function and predicate symbols. In models, a pattern evaluates into a power-set domain (the set of values that *match* it), in contrast to FOL where functions and predicates map into a regular domain. Matching logic uniformly generalizes several logical frameworks important for program analysis, such as: propositional logic, algebraic specification, FOL with equality, modal logic, and separation logic. Patterns can specify separation requirements at any level in any program configuration, not only in the heaps or stores, without any special logical constructs for that: the very nature of pattern matching is that if two structures are matched as part of a pattern, then they can only be spatially separated. Like FOL, matching logic can also be translated into pure predicate logic with equality, at the same time admitting its own sound and complete proof system. A practical aspect of matching logic is that FOL reasoning with equality remains sound, so off-the-shelf provers and SMT solvers can be used for matching logic reasoning. Matching logic is particularly well-suited for reasoning about programs in programming languages that have an operational semantics, but it is not limited to this.

Introduction. In their simplest form, as term templates with variables, patterns abound in mathematics and computer science. They match a concrete, or ground, term if and only if there is some substitution applied to the pattern's variables that makes it equal to the concrete term, possibly via domain reasoning. This means, intuitively, that the concrete term obeys the structure specified by the pattern. We show that when combined with logical connectives and variable constraints and quantifiers, patterns provide a powerful means to specify and reason about the structure of states, or configurations, of a programming language.

Matching logic was born from our belief that programming languages must have formal definitions, and that tools for a given language, such as interpreters, compilers, state-space explorers, model checkers, deductive program verifiers, etc., can be derived from just *one* reference formal definition of the language, which is executable. No other semantics for the same language should be needed.

For example, [3] presents a program verification module of based on matching logic which takes the respective operational semantics of C [4], Java [1], and JavaScript [5] as input and yields automated program verifiers for these languages, capable of verifying challenging heap-manipulating programs at performance comparable to that of state-of-the-art verifiers specifically crafted for those languages.

Matching logic is particularly well-suited for reasoning about programs when their language has an operational semantics. That is because its patterns give us full access to all the details in a program configuration, at the same time allowing us to hide irrelevant detail using existential quantification or separately defined abstractions. Also, both the operational semantics of a language and its reachability properties can be encoded as rules $\varphi \Rightarrow \varphi'$ between patterns, called *reachability rules* in [3,2,7,8], and one generic, language-independent proof system can be used both for executing programs and for proving them correct. In both cases, the operational semantics rules are used to advance the computation. When executing programs the pattern to reduce is ground and the application of the semantic steps becomes conventional term rewriting. When verifying reachability properties, the pattern to reduce is symbolic and typically contains constraints and abstractions, so matching logic reasoning is used in-between semantic rewrite rule applications to re-arrange the configuration so that semantic rules match or assertions can be proved. We refer the interested reader to [3] for full details on our recommended verification approach using matching logic.

Although we favor the verification approach above, which led to the development of matching logic, there is nothing to limit the use of matching logic with other verification approaches, as an intuitive and succinct notation for encoding state properties. For example, one cantake an existing separation logic semantics of a language, regard it as a matching logic semantics and then extend it to also consider structures in the configuration that separation logic was not meant to directly reason about, such as function/exception/break-continue stacks, input/output buffers, etc. For this reason, we here present matching logic as a stand-alone logic, without favoring any particular use of it.

Syntax. Matching logic is a logic centered around the notion of patterns:

Definition 1. *Let (S, Σ) be a many-sorted signature of* ***symbols****. Matching logic (S, Σ)-**formulae**, also called (S, Σ)-**patterns**, or just (matching logic)* ***formulae*** *or* ***patterns*** *when (S, Σ) is understood from context, are inductively defined as follows for all sorts $s \in S$:*

$$\begin{array}{lll}
\varphi_s ::= x \in \mathit{Var}_s & & //\ \textit{Variable} \\
\quad \mid \sigma(\varphi_{s_1}, ..., \varphi_{s_n}) \ \ \textit{with } \sigma \in \Sigma_{s_1 \ldots s_n, s} & (\textit{written } \Sigma_{\lambda,s} \textit{ when } n = 0) & //\ \textit{Structure} \\
\quad \mid \neg \varphi_s & & //\ \textit{Complement} \\
\quad \mid \varphi_s \wedge \varphi_s & & //\ \textit{Intersection} \\
\quad \mid \exists x \,.\, \varphi_s \ \textit{with } x \in \mathit{Var} \quad (\textit{of any sort}) & & //\ \textit{Binding}
\end{array}$$

Let Pattern *be the S-sorted set of patterns. By abuse of language, we refer to the symbols in Σ also as patterns: think of $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ as the pattern $\sigma(x_1 : s_1, \ldots, x_n : s_n)$.*

We argue that the syntax of patterns above is necessary in order to express meaningful patterns, and at the same time it is minimal. Indeed, variable patterns allow us to extract the matched elements or structure and possibly use them in other places in more complex patterns. Forming new patterns from existing patterns by adding more structure/symbols to them is standard and the very basic operation used to construct terms, which are the simplest patterns. Complementing and intersecting patterns allows us to reason with patterns the same way we reason with logical propositions and formulae. Finally, the existential binder serves a dual role. On the one hand, it allows us to abstract away irrelevant parts of the matched structure, which is particularly useful when defining and reasoning about program invariants or structural framing. On the other hand, it allows us to define complex patterns with binders in them, such as λ-, μ-, or ν-bound terms/patterns (to be presented elsewhere).

Semantics. In their simplest form, as terms with variables, patterns are usually matched by other terms that have more structure, possibly by ground terms. However, sometimes we may need to do the matching modulo some background theories or modulo some existing domains, for example integers where addition is commutative or $2 + 3 = 1 + 4$, etc. For maximum generality, we prefer to impose no theoretical restrictions on the models in which patterns are interpreted, or matched, leaving such restrictions to be dealt with in implementations (for example, one may limit to free models, or to ones for which decision procedures exist, etc.).

Definition 2. *A* ***matching logic*** *(S, Σ)**-model** M, or just a Σ**-model** when S is understood, or simply a* ***model*** *when both S and Σ are understood, consists of:*

1. *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*
2. *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.*

Note that symbols are interpreted as relations, and that the usual (S, Σ)-algebra models are a special case of matching logic models, where $|\sigma_M(m_1, \ldots, m_n)| = 1$ for any $m_1 \in M_{s_1}, \ldots, m_n \in M_{s_n}$. Similarly, partial (S, Σ)-algebra models also fall as special case, where $|\sigma_M(m_1, \ldots, m_n)| \leq 1$, since we can capture the undefinedness of σ_M on $m_1, \ldots, m_n$ with $\sigma_M(m_1, \ldots, m_n) = \emptyset$. We tacitly use the same notation σ_M for its extension to argument sets, $\mathcal{P}(M_{s_1}) \times \cdots \times \mathcal{P}(M_{s_n}) \to \mathcal{P}(M_s)$, that is,

$$\sigma_M(A_1, \ldots, A_n) = \bigcup\{\sigma_M(a_1, \ldots, a_n) \mid a_1 \in A_1, \ldots, a_n \in A_n\}$$

where $A_1 \subseteq M_{s_1}, \ldots, A_n \subseteq M_{s_n}$.

Definition 3. *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* $\varphi \in \text{PATTERN}_s$
- $\overline{\rho}(\varphi_1 \wedge \varphi_2) = \overline{\rho}(\varphi_1) \cap \overline{\rho}(\varphi_2)$ *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).*

It is easy to see that the usual notion of term matching is an instance of the above; indeed, if φ is a term with variables and M is the ground term model, then a ground term a matches φ iff there is some substitution ρ such that $\rho(\varphi) = a$. It may be insightful to note that patterns can also be regarded as predicates, when we think of "a matches pattern φ" as "predicate φ holds in a". But matching logic allows more complex patterns than terms or predicates, and models which are not necessarily conventional (term) algebras.

Note that property "if φ closed then $M \models \neg\varphi$ iff $M \not\models \varphi$", which holds in classical logics like FOL, does not hold in matching logic. This is because $M \models \neg\varphi$ means $\neg\varphi$ is matched by all elements, i.e., φ is matched by no element, while $M \not\models \varphi$ means φ is not matched by some elements. These two notions are different when patterns can have more than two interpretations, which happens when M can have more than one element.

Definition 4. *Pattern φ is* ***valid****, written $\models \varphi$, iff $M \models \varphi$ for all M. If $F \subseteq$ PATTERN then $M \models F$ iff $M \models \varphi$ for all $\varphi \in F$. F* ***entails*** *φ, written $F \models \varphi$, iff for each M, $M \models F$ implies $M \models \varphi$. A* ***matching logic specification*** *is a triple (S, Σ, F) with $F \subseteq$ PATTERN.*

The journal paper upon which this presentation is based [6] gives a more detailed introduction to the logic, its relation with existing logics, as well as a specialized (sound and complete) proof system for it.

References

1. Denis Bogdănaş and Grigore Roşu. K-Java: A Complete Semantics of Java. In *POPL'15*, pages 445–456. ACM, January 2015.
2. Andrei Ştefănescu, Ştefan Ciobâcă, Radu Mereuţă, Brandon M. Moore, Traian Florin Şerbănuţă, and Grigore Roşu. All-path reachability logic. In *RTA-TLCA'14*, volume 8560 of *LNCS*, pages 425–440. Springer, 2014.
3. Andrei Ştefănescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore Roşu. Semantics-based program verifiers for all languages. In *OOPSLA'16*, pages 74–91. ACM, 2016.
4. Chris Hathhorn, Chucky Ellison, and Grigore Roşu. Defining the undefinedness of C. In *PLDI'15*, pages 336–345. ACM, 2015.
5. Daejun Park, Andrei Ştefănescu, and Grigore Roşu. KJS: A complete formal semantics of JavaScript. In *PLDI'15*, pages 346–356. ACM, 2015.

6. Grigore Roşu. Matching logic. *Logical Methods in Computer Science*, to appear, 2017.
7. Grigore Roşu, Andrei Ştefănescu, Ştefan Ciobâcă, and Brandon M. Moore. One-path reachability logic. In *LICS'13*, pages 358–367. IEEE, 2013.
8. Grigore Rosu and Andrei Stefanescu. Checking reachability using matching logic. In *OOPSLA'12*, pages 555–574. ACM, 2012.