MATCHING LOGIC

GRIGORE ROŞU

University of Illinois at Urbana-Champaign, USA
e-mail address: grosu@illinois.edu

Abstract. This paper presents *matching logic*, 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.

Contents

1998 ACM Subject Classification: D.2.4 Software/Program Verification; D.3.1 Formal Definitions and Theory; F.3 LOGICS AND MEANINGS OF PROGRAMS; F.4 MATHEMATICAL LOGIC AND FORMAL LANGUAGES .

Key words and phrases: Program logic; First-order logic; Rewriting; Verification.

Extended version of an invited paper at the 26th International Conference on Rewriting Techniques and Applications (RTA'15), June 29 to July 1, 2015, Warsaw, Poland.

The work presented in this paper was supported in part by the Boeing grant on "Formal Analysis Tools for Cyber Security" 2014-2017, the NSF grants CCF-1218605, CCF-1318191 and CCF-1421575, and the DARPA grant under agreement number FA8750-12-C-0284.

1. 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 inspired from the domain of programming language semantics, specifically from attempting to use operational semantics directly for program verification. Recently, operational semantics of several real languages have been proposed, e.g., of C [34, 49], Java [14], JavaScript [13, 71], Python [46, 77], PHP [37], CAML [70], thanks to the development of semantics engineering frameworks like PLT-Redex [54], Ott [87], $\mathbb{K}$ [82, 83], etc., which make defining an operational semantics for a programming language almost as easy as implementing an interpreter, if not easier. Operational semantics are comparatively

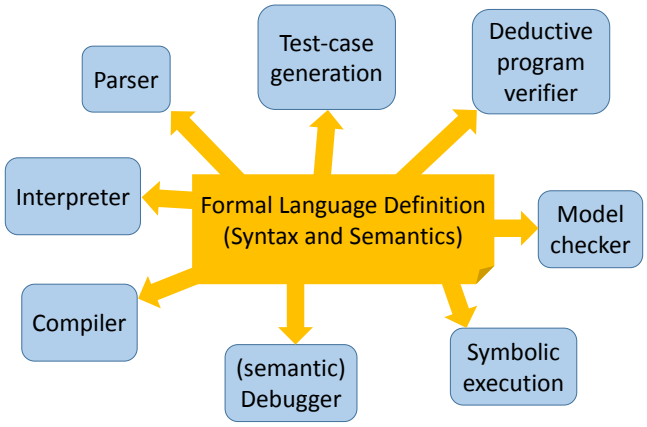

Figure 1: Architecture of the $\mathbb{K}$ framework, powered by matching logic

easy to define and understand, require little formal training, scale up well, and, being executable, can be tested. Indeed, the language semantics above have more than 1,000 (some even more than 3,000) semantic rules and have been tested on benchmarks/test-suites that language implementations use to test their conformance, where available. Thus, operational semantics are typically used as trusted reference models for the defined languages. We would like to use such operational semantics of languages, *unchanged*, for program verification.

Despite their advantages, operational semantics are rarely used directly for program verification, because the general belief is that proofs tend to be low-level, as they work directly with the corresponding transition system. Hoare [50] or dynamic [47] logics are typically used, because they allow higher level reasoning. However, these come at the cost of (re)defining the language semantics as a set of abstract proof rules, which are harder to understand and trust. The state-of-the-art in mechanical program verification is to develop and prove such language-specific proof systems sound w.r.t. a trusted operational semantics [66, 52, 3], but that needs to be done for each language separately and is labor intensive.

Defining even one complete semantics for a real language like C or Java is already a huge effort. Defining multiple semantics, each good for a different purpose, is at best uneconomical, with or without proofs of soundness w.r.t. the reference semantics. It is therefore not surprising that many practical program verifiers forgo defining a semantics altogether, and instead they implement ad-hoc verification condition (VC) generation, sometimes via (unverified) translations to intermediate verification languages like Boogie [4] or Why3 [38]. For example, program verifiers for C like VCC [26] and Frama-C [38], and for Java like jStar [32] take this approach. Also, *none* of the 35 verifiers that participated in the 2016 software verification competition (SV-COMP) [10] appear to be based on a formal semantics of any kind. The consequence is that such tools cannot be trusted. We would like program verifiers, ideally, to produce proof certificates whose trust base is only an operational semantics of the target language, same as mechanical verifiers based on Coq [61] or Isabelle [67] do, but without the effort to define any other semantics of the same language, either directly as a separate proof system or indirectly by extending the operational semantics with language-specific lemmas. We would like program verifiers, ideally, to take an operational semantics of a language as input and to yield, as output, a verifier for that language which is as easy to use and as efficient as verifiers specifically developed for that language.

```
struct listNode { int val; struct listNode *next; };

void list_read_write(int n) {
```
rule $\langle \$ \Rightarrow \texttt{return;}\ \cdots\rangle_{\mathsf{code}}\ \langle \mathsf{A} \Rightarrow \cdot\ \cdots\rangle_{\mathsf{in}}\ \langle \cdots\ \cdot \Rightarrow \mathsf{rev}(\mathsf{A})\rangle_{\mathsf{out}}\ \wedge\ \texttt{n} = \mathsf{len}(\mathsf{A})$
```
  int i=0;
  struct listNode *x=0;
```
inv $\langle \beta\ \wedge\ \mathsf{len}(\beta) = \texttt{n} - \texttt{i}\ \wedge\ \texttt{i} \leq \texttt{n}\ \cdots\rangle_{\mathsf{in}}\ \langle \mathsf{list}(\texttt{x}, \alpha)\ \cdots\rangle_{\mathsf{heap}}\ \wedge\ \mathsf{A} = \mathsf{rev}(\alpha)@\beta$
```
  while (i < n) {
    struct listNode *y = x;
    x = (struct listNode*) malloc(sizeof(struct listNode));
    scanf("%d", &(x->val));
    x->next = y;
    i += 1; }
```
inv $\langle \cdots\ \alpha\rangle_{\mathsf{out}}\ \langle \mathsf{list}(\texttt{x}, \beta)\ \cdots\rangle_{\mathsf{heap}}\ \wedge\ \mathsf{rev}(\mathsf{A}) = \alpha@\beta$
```
  while (x) {
    struct listNode *y;
    y = x->next;
    printf("%d␣",x->val);
    free(x);
    x = y; }
}
```

Figure 2: Reading, storing, and reverse writing a sequence of integers

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. This belief is reflected in the design of the $\mathbb{K}$ framework [82, 83] (`http://kframework.org`), illustrated in Figure 1. This is the ideal scenario and there is enough evidence that it is within our reach in the short term. For example, [28] presents a program verification module of $\mathbb{K}$, based on matching logic, which takes the respective operational semantics of C [49], Java [14], and JavaScript [71] 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. A precursor of this verifier, MatchC [85], has an online interface at `http://matching-logic.org` where one can verify dozens of predefined programs or new ones; e.g., the program in Figure 2 is under the `io` folder and it takes about 150ms to verify.

To reason about programs we need to be able to reason about program *configurations*. Specifically, we need to define configuration abstractions and reason with them. Consider, for example, the program in Figure 2 which shows a C function that reads `n` elements from the standard input and prints them to the standard output in reversed order (for now, we can ignore the specifications, which are grayed). While doing so, it allocates a singly linked list storing the elements as they are read, and then deallocates the list as the elements are printed. In the end, the heap stays unchanged. To state the specification of this program, we need to match an abstract sequence of `n` elements in the input buffer, and then to match its reverse at the end of the output buffer when the function terminates. Further, to state

the invariants of the two loops we need to identify a singly linked pattern in the heap, which is a partial map. Many such sequence or map patterns, as well as operations on them, can be defined using conventional algebraic data types (ADTs). But some of them cannot.

A major limitation of ADTs and of first-order logic (FOL) is that operation symbols are interpreted as functions in models, which sometimes is insufficient. E.g., a two-element linked list in the heap (we regard heaps as maps from natural number locations to values) starting with location 7 and holding values 9 and 5, written as $\mathit{list}(7, 9@5)$, can allow infinitely many heap values, one for each location where the value 5 may be stored. So we cannot define *list* as an operation symbol $\mathit{Int} \times \mathit{Seq} \to \mathit{Map}$. The FOL alternative is to define *list* as a predicate $\mathit{Int} \times \mathit{Seq} \times \mathit{Map}$, but mentioning the map all the time as an argument makes specifications verbose and hard to read, use and reason about. An alternative, proposed by separation logic [78], is to fix and move the map domain from explicit in models to implicit in the logic, so that $\mathit{list}(7, 9@5)$ is interpreted as a predicate but the non-deterministic map choices are implicit in the logic. We then may need custom separation logics for different languages that require different variations of map models or different configurations making use of different kinds of resources. This may also require specialized separation logic provers needed for each, or otherwise encodings that need to be proved correct. Finally, since the map domain is not available as data, one cannot use FOL variables to range over maps and thus proof rules like "heap framing" need to be added to the logic explicitly.

Matching logic avoids the limitations of both approaches above, by interpreting its terms/formulae as *sets* of values. Matching logic's formulae, called *patterns*, are built using variables, symbols from a signature, and FOL connectives and quantifiers. We can think of matching logic as collapsing the function and predicate symbols of FOL, allowing patterns to be simultaneously regarded *both* as terms and as predicates. When regarded as terms they build structure, when regarded as predicates they express constraints. Semantically, the matching logic models are similar to the FOL models, except that the symbols in the signature are interpreted as functions returning sets of values instead of single values. Patterns are then also interpreted as sets of values, where conjunction is interpreted as intersection, negation as complement, and the existential quantifier as union over all compatible valuations. The name "matching logic" was inspired from the case when the model is that of terms, common in the context of language semantics, where terms represent (fragments of) program configurations. There, a pattern is interpreted as the set of terms that match it.

The (grayed) specifications in Figure 2 show examples of matching logic patterns, over the signature used to define the semantics of C [49]. The signature includes symbols corresponding to the syntax of the language, to semantic constructs such as $\langle \_ \rangle_{\mathsf{code}}$ holding the remaining code fragment, $\langle \_ \rangle_{\mathsf{heap}}$ holding the current heap as a map, and $\langle \_ \rangle_{\mathsf{in}}$ and $\langle \_ \rangle_{\mathsf{out}}$ holding the current input and resp. output buffers as sequences, among many others. Let us discuss the invariant pattern of the first loop (second grayed area). It says that the pattern $\mathsf{list}(\mathsf{x}, \alpha)$ is matched somewhere in the heap, and that the sequence β of size $\mathsf{n} - \mathsf{i}$ is available at the beginning of the input buffer such that A is the reverse of the sequence that x points to, $\mathsf{rev}(\alpha)$, concatenated with β. The ellipses "$\cdots$" are syntactic sugar for existentially quantified variables, which we call "structural frame variables". Note how symbols from the signature are mixed with logical constructs, and how variables can range over any data stored in configurations, including over heap fragments. In addition to the implicit existential quantifiers for "$\cdots$", the sequence β under the $\langle \_ \rangle_{\mathsf{in}}$ symbol is conjuncted with logical constraints about its length; also, the pair consisting of the $\langle \_ \rangle_{\mathsf{in}}$ and $\langle \_ \rangle_{\mathsf{heap}}$ patterns at the top, which is itself a configuration pattern, is conjuncted with the equality constraint

$\mathsf{A} = \mathsf{rev}(\alpha)@\beta$. While such mixes of symbols and logical connectives are disallowed in other logics, such as FOL or separation logic, they are not only well-formed but also strongly encouraged to be used in matching logic; besides succinctness of specifications, they also allow for local reasoning. This is discussed in detail shortly, in the example in Section 2.2.

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 (e.g., the "..." framing variables in Figure 2) or separately defined abstractions (e.g., the $\mathsf{list}(\mathsf{x}, \alpha)$ pattern in Figure 2). 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 [28, 27, 80, 85], 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 [28] 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, Proposition 9.2 tells us that any separation logic formula is a matching logic pattern *as is*. So one can, for example, take 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.

This paper is an extended version of the RTA'15 conference paper [79], which was the first to allow the unrestricted mix of symbols and logical quantifiers in patterns. A much simpler variant of matching logic was introduced in 2010 in [81] as a state specification logic, and has been used since then in several verification efforts [84, 85, 86, 80, 27, 28], and implemented in MatchC [85] by reduction to Maude [25] (for matching) and to Z3 [29] (for domain reasoning). However, that matching logic variant shares only the basic intuition of "terms as formulae" with the logic presented in this paper, and was only syntactic sugar for first-order logic (FOL) with equality in a fixed model, essentially allowing only term patterns t and regarding them as syntactic sugar for equalities $\Box = t$ (see Section 12).

Section 2 introduces the syntax and semantics of matching logic, as well as some basic properties. Sections 3 and 4 show how propositional calculus and, respectively, pure predicate logic fall as instances of matching logic. Section 5 shows how several important mathematical concepts can be defined in matching logic, such as definedness, equality, membership, and functions. Using these, Sections 6, 7, 8 and 9 then show how algebraic specifications, first-order logic, modal logic and, respectively, separation logic also fall as instances of matching logic. Section 10 shows that, like FOL, matching logic also reduces to pure predicate logic with equality. Section 11 introduces our sound and complete proof system for matching logic. Section 12 discusses related work and Section 13 concludes.

2. Matching Logic: Basic Notions

We assume the reader is familiar with many-sorted sets, functions, and first-order logic (FOL). For any given set of sorts S, we assume Var is an S-sorted set of variables, sortwise infinite and disjoint. We may write $x : s$ instead of $x \in Var_s$, and when the sort of x is irrelevant we just write $x \in Var$. We let $\mathcal{P}(M)$ denote the powerset of a many-sorted set M, which is itself many-sorted. We only treat the many-sorted case here, but we see no inherent limitations in extending the constructions and results in this paper to the order-sorted case.

2.1. **Patterns.** We start by defining the syntax of patterns.

Definition 2.1. *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 $s \in S$:*

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

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).

To ease notation, $\varphi \in$ Pattern means φ is a pattern, while $\varphi_s \in$ Pattern or $\varphi \in$ Pattern$_s$ that it has sort s. We adopt the following derived constructs ("syntactic sugar"):

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

Intuitively, $\top$ is a pattern that is matched by all elements, $\bot$ is matched by no elements, $\varphi_1 \vee \varphi_2$ is matched by all elements matching φ_1 or φ_2, and so on. We will shortly formalize this intuition. We assume the usual precedence of the FOL-like constructs, with $\neg$ binding tighter than $\wedge$ tighter than $\vee$ tighter than $\to$ tighter than $\leftrightarrow$ tighter than the quantifiers.

We adapt from first-order logic the notions of free variable, (variable capture free) substitution, and variable renaming, briefly recalled below. Let $FV(\varphi)$ denote the *free variables* of φ, defined as follows: $FV(x) = \{x\}$, $FV(\sigma(\varphi_{s_1}, ..., \varphi_{s_n})) = FV(\varphi_{s_1}) \cup \cdots \cup FV(\varphi_{s_n})$, $FV(\neg \varphi) = FV(\varphi)$, $FV(\varphi_1 \wedge \varphi_2) = FV(\varphi_1) \cup FV(\varphi_2)$, and $FV(\exists x . \varphi) = FV(\varphi) \setminus \{x\}$. Similarly, the usual variable capture free *substitution*: $x[\varphi/x] = \varphi$ and $y[\varphi/x] = y$ when variable

y is different from x, $\sigma(\varphi_{s_1}, ..., \varphi_{s_n})[\varphi/x] = \sigma(\varphi_{s_1}[\varphi/x], ..., \varphi_{s_n}[\varphi/x])$, $(\varphi_1 \wedge \varphi_2)[\varphi/x] = \varphi_1[\varphi/x] \wedge \varphi_2[\varphi/x]$, $(\neg\varphi')[\varphi/x] = \neg(\varphi'[\varphi/x])$, and $(\exists x.\varphi')[\varphi/x] = \exists x.\varphi'$ and $(\exists y.\varphi')[\varphi/x] = \exists y.(\varphi'[\varphi/x])$ when variable y is different from x and $y \notin FV(\varphi)$ (to avoid variable capture). And *variable renaming*, $\exists x.\varphi \equiv \exists y.(\varphi[y/x])$, which can be used to avoid variable capture.

2.2. **Example.** There are many examples of patterns throughout the paper resulting from formulae in various other logics that are captured by matching logic, such as propositional logic (Section 3), predicate logic (Section 4), algebraic specifications (Section 6) first-order logic (Section 7) modal logic (Section 8), and separation logic (Section 9). We will discuss them in their respective sections, showing that formulae in these logics can be regarded as matching logic patterns. Here, instead, we discuss an example inspired from programming language semantics, which is the area that motivated the development of matching logic.

Consider the operational semantics of a real language like C, whose configuration has more than 100 semantic components [34, 49, 28]. The semantic components, here called "cells" and written using symbols $\langle ... \rangle_{\mathsf{cell}}$, can be nested and their grouping (symbol) is governed by associativity and commutativity axioms. There is a top cell $\langle ... \rangle_{\mathsf{cfg}}$ holding subcells $\langle ... \rangle_{\mathsf{code}}$, $\langle ... \rangle_{\mathsf{heap}}$, $\langle ... \rangle_{\mathsf{in}}$, $\langle ... \rangle_{\mathsf{out}}$ among many others, holding the current code fragment, heap, input buffer, output buffer, respectively. We cannot show the signature of all the symbols defining the configuration of a language like C for space reasons, but encourage the interested reader to check the aforementioned papers. We only show a small subset of symbols that is sufficient to write interesting patterns for illustration purposes, mentioning that nothing changes in the subsequent developments of matching logic as the signature grows or changes. That is, we do not have a matching logic for C, another for Java, another for JavaScript, etc.; all these languages have their respective signatures and patterns, and the same matching logic machinery applies to all of them in the same way.

To motivate certain patterns below, we will refer to results that are introduced later in the paper. The purpose of this example, however, is to illustrate and discuss various kinds of patterns, and especially to show that it is useful to mix symbols with logical connectives. The hasty reader can only skim the patterns and their descriptions below for now, and revisit the example later as other results back-reference it.

Consider the signature (S, Σ) in Figure 3, consisting of symbols needed to construct semantic configurations for a C-like language. Usual terms are already patterns, in particular the first while loop in the program in Figure 2, say `LOOP`. So are terms with variables, e.g.:

$$\langle \langle \texttt{LOOP}\ k \rangle_{\mathsf{code}}\ \langle \texttt{x} \mapsto x,\ \texttt{n} \mapsto n,\ \texttt{i} \mapsto i,\ e \rangle_{\mathsf{env}}\ \langle x \mapsto a,\ x+1 \mapsto y,\ h \rangle_{\mathsf{heap}}\ \langle \beta \rangle_{\mathsf{in}}\ \langle \epsilon \rangle_{\mathsf{out}} \rangle_{\mathsf{cfg}}$$

The intuition for this pattern is that it matches all the configurations whose code starts with `LOOP` (k, the "code frame", matches the rest of the code), whose environment binds program identifiers `n` and `i` to values n and i, respectively, and `x` to location x (e, the "environment frame", matches the rest of the environment map) such that both x and $x+1$ are allocated and bound to some values in the heap (h, the "heap frame", matches the rest of the heap), whose input buffer contains some sequence (β) and whose output buffer contains the empty sequence. This intuition will be formalized shortly. Also, we will show how the various symbols can be constrained or defined axiomatically, like in algebraic (Section 6) or FOL (Section 7) specifications; for example, sequences are associative and have ϵ as unit, maps and *Cfg* are both associative and commutative with "." as unit, $\mathsf{len}(i@\alpha) = 1 + \mathsf{len}(\alpha)$, etc.

The interesting patterns are those combining symbols and logical connectives. For example, suppose that we want to restrict the pattern above to only match configurations

$$
\begin{array}{rcll}
S & = & \{ \ \mathit{Id}, \mathit{Exp}, \mathit{Stmt}, ... & // \text{ code synactic categories} \\
 & & \ \ \mathit{Bool}, \mathit{Nat}, \mathit{Int}, ... & // \text{ basic domains} \\
 & & \ \ \mathit{Seq}_{\mathit{Int}}, \mathit{Map}_{\mathit{Id,Nat}}, \mathit{Map}_{\mathit{Nat,Int}}, ... & // \text{ more domains} \\
 & & \ \ \mathit{CfgCell}, \mathit{Cfg}, & // \text{ top cell and contents} \\
 & & \ \ \mathit{CodeCell}, \mathit{EnvCell}, \mathit{HeapCell}, \mathit{InCell}, \mathit{OutCell}, & // \text{ config cells} \\
 & & ... \ \ \} & \\
 & & \Sigma_{\mathit{Id\,Exp},\,\mathit{Stmt}} = \{\texttt{\_=\_;}, \ ...\} & // \text{ assignment, ...} \\
 & & \Sigma_{\mathit{Exp\,Stmt\,Stmt},\,\mathit{Stmt}} = \{\texttt{if(\_)\{\_\}else\{\_\}}, \ ...\} & // \text{ conditional, ...} \\
 & & \Sigma_{\mathit{Exp\,Stmt},\,\mathit{Stmt}} = \{\texttt{while(\_)\{\_\}}, \ ...\} & // \text{ while loop, ...} \\
 & & \Sigma_{\mathit{Stmt\,Stmt},\,\mathit{Stmt}} = \{ \ \texttt{\_\_} \ \} & // \text{ sequential composition} \\
... & // \text{ other syntactic language constructs} & & \\
 & & \Sigma_{\mathit{Stmt},\,\mathit{Cfg}} = \{ \ \langle \texttt{\_} \rangle_{\mathsf{code}} \ \} & // \text{ cell holding the code} \\
 & & \Sigma_{\lambda,\,\mathit{Map}_{\mathit{Id,Nat}}} = \{ \ . \ \} & // \text{ empty environment map} \\
 & & \Sigma_{\mathit{Id\,Nat},\mathit{Map}_{\mathit{Id,Nat}}} = \{ \ \texttt{\_} \mapsto \texttt{\_} \ \} & // \text{ one-binding environment map} \\
 & & \Sigma_{\mathit{Map}_{\mathit{Id,Nat}}\,\mathit{Map}_{\mathit{Id,Nat}},\,\mathit{Map}_{\mathit{Id,Nat}}} = \{ \ \texttt{\_,\_} \ \} & // \text{ environment map merge} \\
 & & \Sigma_{\mathit{Map}_{\mathit{Id,Nat}},\,\mathit{Cfg}} = \{ \ \langle \texttt{\_} \rangle_{\mathsf{env}} \ \} & // \text{ cell holding the environment map} \\
... & \mathit{Map}_{\mathit{Nat,Int}} \text{ symbols defined similarly to } \mathit{Map}_{\mathit{Id,Nat}} & & \\
 & & \Sigma_{\mathit{Map}_{\mathit{Nat,Int}},\,\mathit{Cfg}} = \{ \ \langle \texttt{\_} \rangle_{\mathsf{heap}} \ \} & // \text{ cell holding the heap map} \\
 & & \Sigma_{\lambda,\,\mathit{Seq}_{\mathit{Int}}} = \{ \ \epsilon \ \} & // \text{ empty sequence} \\
 & & \Sigma_{\mathit{Int},\,\mathit{Seq}_{\mathit{Int}}} = \{ \ \texttt{\_} \ \} & // \text{ one-integer sequence} \\
 & & \Sigma_{\mathit{Seq}_{\mathit{Int}}\,\mathit{Seq}_{\mathit{Int}},\,\mathit{Seq}_{\mathit{Int}}} = \{ \ \texttt{\_@\_} \ \} & // \text{ sequence concatenation} \\
 & & \Sigma_{\mathit{Seq}_{\mathit{Int}},\,\mathit{Nat}} = \{ \ \mathsf{len} \ \} & // \text{ sequence length} \\
 & & \Sigma_{\mathit{Seq}_{\mathit{Int}},\,\mathit{Seq}_{\mathit{Int}}} = \{ \ \mathsf{rev} \ \} & // \text{ sequence reverse} \\
 & & \Sigma_{\mathit{Seq}_{\mathit{Int}},\,\mathit{Cfg}} = \{ \ \langle \texttt{\_} \rangle_{\mathsf{in}} \ \} & // \text{ input buffer} \\
 & & \Sigma_{\mathit{Seq}_{\mathit{Int}},\,\mathit{Cfg}} = \{ \ \langle \texttt{\_} \rangle_{\mathsf{out}} \ \} & // \text{ output buffer} \\
 & & \Sigma_{\lambda,\,\mathit{Cfg}} = \{ \ . \ \} & // \text{ empty configuration contents} \\
 & & \Sigma_{\mathit{Cfg},\,\mathit{Cfg}} = \{ \ \texttt{\_\_} \ \} & // \text{ merging configuration contents} \\
 & & \Sigma_{\mathit{Cfg},\,\mathit{CfgCell}} = \{ \ \langle \texttt{\_} \rangle_{\mathsf{cfg}} \ \} & // \text{ top configuration cell}
\end{array}
$$

Figure 3: Signature for building program configurations in a C-like language

where $i \leq n$. As discussed later in the paper (Section 5.2), equality can be axiomatized in matching logic and used in any sort context. Also, due to their ubiquity, Boolean expressions are allowed to be used in any sort context unchanged, with the meaning that they equal *true*; that is, we write just b instead of $b = \mathit{true}$ (Section 5.8). With these, we can restrict the pattern above as follows (note the top-level conjuction):

$$\langle\langle \texttt{LOOP}\ k\rangle_{\mathsf{code}}\ \langle \texttt{x} \mapsto x,\ \texttt{n} \mapsto n,\ \texttt{i} \mapsto i,\ e\rangle_{\mathsf{env}}\ \langle x \mapsto a,\ x+1 \mapsto y,\ h\rangle_{\mathsf{heap}}\ \langle\beta\rangle_{\mathsf{in}}\ \langle\epsilon\rangle_{\mathsf{out}}\rangle_{\mathsf{cfg}}\ \wedge\ i \leq n$$

Quantifiers can be used, for example, to abstract away irrelevant parts of the pattern. Suppose, for example, that we work in a context where the code and the output cells are irrelevant, and so are the frames of the environment and heap cells. Then we can "hide" them to the context as follows:

$$(\exists c\,.\,\exists e\,.\,\exists h\,.\,\langle\langle \texttt{x} \mapsto x,\ \texttt{n} \mapsto n,\ \texttt{i} \mapsto i,\ e\rangle_{\mathsf{env}}\ \langle x \mapsto a,\ x+1 \mapsto y,\ h\rangle_{\mathsf{heap}}\ \langle\beta\rangle_{\mathsf{in}}\ c\rangle_{\mathsf{cfg}})\ \wedge\ i \leq n$$

Following a notational convention proposed and implemented in $\mathbb{K}$ (`http://kframework.org` [82, 83]), we use "..." as syntactic sugar for such existential quantifiers used for framing:

$$\langle\langle \mathtt{x} \mapsto x,\ \mathtt{n} \mapsto n,\ \mathtt{i} \mapsto i\ \ \cdots\rangle_{\mathsf{env}}\ \langle x \mapsto a,\ x+1 \mapsto y\ \ \cdots\rangle_{\mathsf{heap}}\ \langle\beta\rangle_{\mathsf{in}}\ \ \cdots\rangle_{\mathsf{cfg}}\ \wedge\ i \leq n$$

It is often the case that program identifiers are bound to default mathematical variables (their symbolic values) in the environment, and then the mathematical variables are used in many other parts of the configuration pattern to state additional logical or structural constraints. For that reason, we typically want to match the program identifiers to their (symbolic) values once and for all with a separate, default (sub)pattern, which is then not mentioned anymore in subsequent patterns:

$$\begin{array}{ll} \langle\langle \mathtt{x} \mapsto x,\ \mathtt{n} \mapsto n,\ \mathtt{i} \mapsto i\ \ \cdots\rangle_{\mathsf{env}}\ \ \cdots\rangle_{\mathsf{cfg}} & \text{// assumed by default below} \\ \wedge\ \langle\langle x \mapsto a,\ x+1 \mapsto y\ \ \cdots\rangle_{\mathsf{heap}}\ \langle\beta\rangle_{\mathsf{in}}\ \ \cdots\rangle_{\mathsf{cfg}}\ \wedge\ i \leq n & \end{array}$$

Note that the pattern above contains two (top-level) $\langle ... \rangle_{\mathsf{cfg}}$ sub-pattern constraints and one logical constraint, $i \leq n$. This pattern will be matched by precisely those configurations that match both sub-patterns and satisfy the constraint, which are the same configurations that match the previous pattern. Therefore, the last two patterns are equal (pattern equality is formalized in Section 5.2; see Notation 5.8 and Proposition 5.9)).

Now suppose that we want to state that location x in the heap points to a linked list over the list data-structure in the program in Figure 2, which comprises a mathematical sequence of integers α. The precise locations of the various nodes in the list are irrelevant. Such a linked-list pattern can be defined by adding a symbol representing it to the signature, say $\mathit{list} \in \Sigma_{\mathit{Nat}\,\mathit{Seq}_{\mathit{Int}},\mathit{Map}_{\mathit{Nat},\mathit{Int}}}$, together with two axioms (similar to those in separation logic, Section 9); it is shown in Section 9.2 that the pattern $\mathit{list}(x,\alpha)$ is matched by precisely all the (infinitely many) linked lists starting with location x and containing the sequence of elements α. This shows why we want pattern symbols to be interpreted into power-set domains, so they can evaluate to sets of elements (all those that match them) instead of just elements. Matching logic also allows us to axiomatically state that a symbol is to be interpreted as a function (Section 5.4); in fact, in this simple example we assume all the symbols of our signature Σ above to be constrained to be functions, except for those of *Map* results. We can now refine the pattern above as follows:

$$\langle\langle \mathit{list}(x,\alpha)\ \ \cdots\rangle_{\mathsf{heap}}\ \langle\beta\rangle_{\mathsf{in}}\ \ \cdots\rangle_{\mathsf{cfg}}\ \wedge\ i \leq n$$

Inspired from the invariant of the first loop in Figure 2, let us add some more constraints:

$$\langle\langle \mathit{list}(x,\alpha)\ \ \cdots\rangle_{\mathsf{heap}}\ \langle\beta\ \ \cdots\rangle_{\mathsf{in}}\ \ \cdots\rangle_{\mathsf{cfg}}\ \wedge\ \mathsf{len}(\beta) = n - i\ \wedge\ i \leq n\ \wedge\ A = \mathsf{rev}(\alpha)@\beta$$

The pattern above is additionally stating that the $\langle ... \rangle_{\mathsf{in}}$ cell starts with a prefix of size equal to $n - i$ which appended to the reverse of the sequence that x points to in the heap equals the original input sequence A. We can arrange the pattern to better localize the logical constraints to the sub-patterns for which they are relevant. For example, the first two constraints are relevant for the sequence β, so we can move them to their place:

$$\langle\langle \mathit{list}(x,\alpha)\ \ \cdots\rangle_{\mathsf{heap}}\ \langle\beta\ \wedge\ \mathsf{len}(\beta) = n - i\ \wedge\ i \leq n\ \ \cdots\rangle_{\mathsf{in}}\ \ \cdots\rangle_{\mathsf{cfg}}\ \wedge\ A = \mathsf{rev}(\alpha)@\beta$$

The above transformation is indeed correct, thanks to Proposition 5.12 (constraint propagation). Similarly, the remaining constraint can be localized to the two cells that need it. Using also the fact that cell concatenation is commutative, we rewrite the pattern into:

$$\langle\langle\beta\ \wedge\ \mathsf{len}(\beta) = n - i\ \wedge\ i \leq n\ \ \cdots\rangle_{\mathsf{in}}\ \langle \mathit{list}(x,\alpha)\ \ \cdots\rangle_{\mathsf{heap}}\ \wedge\ A = \mathsf{rev}(\alpha)@\beta\ \ \cdots\rangle_{\mathsf{cfg}}$$

The pattern above is very similar to the first invariant in Figure 2; the latter does not mention the top $\langle ... \rangle_{\mathsf{cfg}}$ cell because our implementation adds it automatically. The top cell is not necessary anyway, we added it mostly for uniformity in our notation for configurations.

So constraints can be propagated up and down a pattern to where they are needed. But how are the constraints generated? One way to generate constraints is through reasoning using language semantic rules, such as the case analysis and consequence rules in [28]. Another way to generate constraints is by local reasoning about patterns. For example, using the axioms of *list* in Section 9.2, we can infer $\mathit{list}(x,\alpha) \rightarrow \varphi_1 \vee \varphi_2$, where φ_1 is $\cdot \wedge (\alpha = \epsilon)$ (the empty map "$\cdot$" with constraint "α is the empty sequence") and φ_2 is $\exists a\,.\,\exists \gamma\,.\,\alpha = a@\gamma \wedge \exists y\,.\,(x \mapsto a,\, x+1 \mapsto y,\, \mathit{list}(y,\gamma))$. By Proposition 2.10 (structural framing) and propositional reasoning we can then infer the following pattern:

$$\begin{array}{rl} & \langle\langle \beta \ \wedge\ \mathsf{len}(\beta) = n - i \ \wedge\ i \leq n \ \cdots\rangle_{\mathsf{in}}\ \langle \mathit{list}(x,\alpha) \ \cdots\rangle_{\mathsf{heap}} \ \wedge\ A = \mathsf{rev}(\alpha)@\beta \ \cdots\rangle_{\mathsf{cfg}} \\ \rightarrow & \langle\langle \beta \ \wedge\ \mathsf{len}(\beta) = n - i \ \wedge\ i \leq n \ \cdots\rangle_{\mathsf{in}}\ \langle \varphi_1 \vee \varphi_2 \ \cdots\rangle_{\mathsf{heap}} \ \wedge\ A = \mathsf{rev}(\alpha)@\beta \ \cdots\rangle_{\mathsf{cfg}} \end{array}$$

Since symbol application distributes over $\vee$ (Proposition 2.11), the pattern to the right of $\rightarrow$ above becomes (again via propositional reasoning):

$$\begin{array}{rl} & \langle\langle \beta \ \wedge\ \mathsf{len}(\beta) = n - i \ \wedge\ i \leq n \ \cdots\rangle_{\mathsf{in}}\ \langle \varphi_1 \ \cdots\rangle_{\mathsf{heap}} \ \wedge\ A = \mathsf{rev}(\alpha)@\beta \ \cdots\rangle_{\mathsf{cfg}} \\ \vee & \langle\langle \beta \ \wedge\ \mathsf{len}(\beta) = n - i \ \wedge\ i \leq n \ \cdots\rangle_{\mathsf{in}}\ \langle \varphi_2 \ \cdots\rangle_{\mathsf{heap}} \ \wedge\ A = \mathsf{rev}(\alpha)@\beta \ \cdots\rangle_{\mathsf{cfg}} \end{array}$$

We can now propagate the constraints of each of φ_1 and φ_2 up into their respective disjunct above, to be used in combination with the other constrains on sequences.

We stop here with our example. Note that we made no effort above to construct a signature that does not allow junk configurations (for example, there is nothing to stop us from adding two or more heaps in a configuration); such junk configurations can be dismissed either by adding stronger sorting or by well-formedness predicates/patterns. Also, our syntax for empty maps ("$\cdot$") and for map merging ("$\_,\_$") above is different from that in Section 9.2. The syntax above is close to the one we use in our $\mathbb{K}$ implementation, while the syntax in Section 9.2 was specifically chosen to be similar to that of separation logic in order to support the subsequent results in Section 9.3.

2.3. **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.). This has the additional benefit that it yields complete deduction (Section 11).

Definition 2.2. *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* $\sigma_M : M_{s_1} \times \cdots \times M_{s_n} \rightarrow \mathcal{P}(M_s)$ *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 $m_1 \in M_{s_1}$,

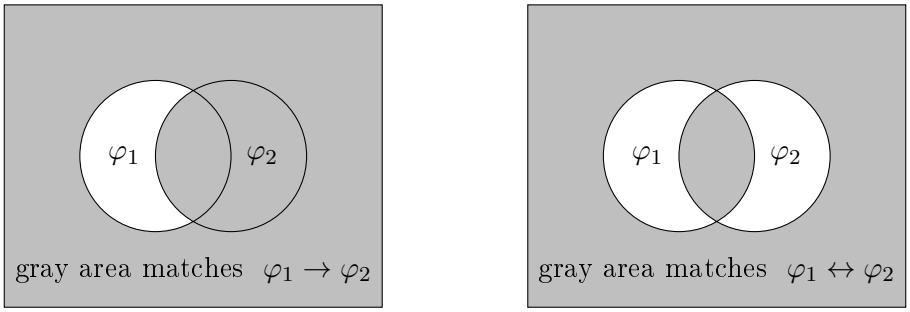

Figure 4: Matching logic semantics of pattern implication and equivalence

$\ldots$, $m_n \in M_{s_n}$. 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 2.3. *Given a model M and a map $\rho : Var \to M$, called an M-**valuation**, let its extension $\overline{\rho} : \text{PATTERN} \to \mathcal{P}(M)$ 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*
- $\overline{\rho}(\exists x.\varphi) = \bigcup\{\overline{\rho'}(\varphi) \mid \rho' : Var \to M,\ \rho'|_{Var\setminus\{x\}} = \rho|_{Var\setminus\{x\}}\} = \bigcup_{a\in M} \overline{\rho[a/x]}(\varphi)$

*where "$\setminus$" is set difference, "$\rho|_V$" 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.

The extension of ρ works as expected with the derived constructs:

- $\overline{\rho}(\top_s) = M_s$ and $\overline{\rho}(\bot_s) = \emptyset$
- $\overline{\rho}(\varphi_1 \vee \varphi_2) = \overline{\rho}(\varphi_1) \cup \overline{\rho}(\varphi_2)$
- $\overline{\rho}(\varphi_1 \to \varphi_2) = \{m \in M_s \mid m \in \overline{\rho}(\varphi_1) \text{ implies } m \in \overline{\rho}(\varphi_2)\} = M_s \setminus (\overline{\rho}(\varphi_1) \setminus \overline{\rho}(\varphi_2))$
- $\overline{\rho}(\varphi_1 \leftrightarrow \varphi_2) = \{m \in M_s \mid m \in \overline{\rho}(\varphi_1) \text{ iff } m \in \overline{\rho}(\varphi_2)\} = M_s \setminus (\overline{\rho}(\varphi_1) \,\Delta\, \overline{\rho}(\varphi_2))$
  ("Δ" is the set symmetric difference operation)
- $\overline{\rho}(\forall x.\varphi) = \bigcap\{\overline{\rho'}(\varphi) \mid \rho' : Var \to M,\ \rho'|_{Var\setminus\{x\}} = \rho|_{Var\setminus\{x\}}\} = \bigcap_{a\in M} \overline{\rho[a/x]}(\varphi)$

Interpreting formulae as sets of elements in models is reminiscent of modal logic, where they are interpreted as the "worlds" in which they hold, and of separation logic, where they are interpreted as the "heaps" they match. We discuss the relationship between matching logic and these logics in depth in Sections 8 and, respectively, 9.

Therefore, the matching logic interpretation of the logical connectives is not two-valued like in classical logics. In particular, the interpretation of $\varphi_1 \to \varphi_2$ is the set of all the elements that if matched by φ_1 then are also matched by φ_2. One should be careful when reasoning with such non-classical logics, as basic intuitions may deceive. For example, the interpretation of $\varphi_1 \to \varphi_2$ is the total set (i.e., same as $\top$) iff all elements matching φ_1 also match φ_2, but it is the empty set iff φ_2 is matched by no elements (same as $\bot$) while φ_1 is matched by all elements (same as $\top$). If in doubt, thanks to the set-theoretical interpretation of the matching logic connectives, we can always draw diagrams to enhance our intuition; for example, Figure 4 depicts the semantics of pattern implication and of pattern equivalence.

When doing logical reasoning with patterns, we sometimes want to think of a pattern exclusively as a "predicate", that is, as something which is either true or false. To avoid using quotes in such situations, we introduce the following:

Definition 2.4. *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.*

Note that $\top_s$ 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. Section 5 will introduce several useful predicate constructs.

Definition 2.5. *M **satisfies** φ_s, written $M \models \varphi_s$, iff $\overline{\rho}(\varphi_s) = M_s$ for all $\rho : Var \to M$.*

Proposition 2.6. *Unless otherwise stated, assume the default pattern sort to be s. Then:*

(1) *If $\rho_1, \rho_2 : Var \to M$, $\rho_1{\restriction}_{FV(\varphi)} = \rho_2{\restriction}_{FV(\varphi)}$ then $\overline{\rho_1}(\varphi) = \overline{\rho_2}(\varphi)$*
(2) *If $x \in Var_s$ then $M \models x$ iff $|M_s| = 1$*
(3) *If $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ and $\varphi_1, \ldots, \varphi_n$ are patterns of sorts $s_1, \ldots, s_n$, respectively, then we have $M \models \sigma(\varphi_1, \ldots, \varphi_n)$ iff $\sigma_M(\overline{\rho}(\varphi_1), \ldots, \overline{\rho}(\varphi_n)) = M_s$ for any $\rho : Var \to M$*
(4) *$M \models \neg\varphi$ iff $\overline{\rho}(\varphi) = \emptyset$ for any $\rho : Var \to M$*
(5) *$M \models \varphi_1 \wedge \varphi_2$ iff $M \models \varphi_1$ and $M \models \varphi_2$*
(6) *If $\exists x.\varphi_s$ closed, $M \models \exists x.\varphi_s$ iff $\bigcup\{\overline{\rho}(\varphi_s) \mid \rho : Var \to M\} = M_s$; hence, $M \models \exists x.x$*
(7) *$M \models \varphi_1 \to \varphi_2$ iff $\overline{\rho}(\varphi_1) \subseteq \overline{\rho}(\varphi_2)$ for all $\rho : Var \to M$*
(8) *$M \models \varphi_1 \leftrightarrow \varphi_2$ iff $\overline{\rho}(\varphi_1) = \overline{\rho}(\varphi_2)$ for all $\rho : Var \to M$*
(9) *$M \models \forall x.\varphi$ iff $M \models \varphi$*

Proof. The proof of each of the properties is below:

(1) Structural induction on φ. The only interesting case is when φ has the form $\exists x.\varphi'$, so $FV(\varphi) = FV(\varphi') \setminus \{x\}$. Then

$$\begin{array}{rcl}
\overline{\rho_1}(\exists x.\varphi') & = & \bigcup\{\overline{\rho_1'}(\varphi') \mid \rho_1' : Var \to M,\ \rho_1'{\restriction}_{Var\setminus\{x\}} = \rho_1{\restriction}_{Var\setminus\{x\}}\} \\
& & \text{(by Definition 2.3)} \\
& = & \bigcup\{\overline{\rho_1'}(\varphi') \mid \rho_1' : Var \to M,\ \rho_1'{\restriction}_{FV(\varphi)} = \rho_1{\restriction}_{FV(\varphi)}\} \\
& & \text{(by the induction hypothesis)} \\
& = & \bigcup\{\overline{\rho_2'}(\varphi') \mid \rho_2' : Var \to M,\ \rho_2'{\restriction}_{FV(\varphi)} = \rho_2{\restriction}_{FV(\varphi)}\} \\
& & \text{(since } \rho_1{\restriction}_{FV(\varphi)} = \rho_2{\restriction}_{FV(\varphi)}\text{)} \\
& = & \bigcup\{\overline{\rho_2'}(\varphi') \mid \rho_2' : Var \to M,\ \rho_2'{\restriction}_{Var\setminus\{x\}} = \rho_2{\restriction}_{Var\setminus\{x\}}\} \\
& & \text{(by the induction hypothesis)} \\
& = & \overline{\rho_2}(\exists x.\varphi')
\end{array}$$

(2) $M \models x$ iff $\overline{\rho}(x) = M_s$ for all $\rho : Var \to M$, iff $\{\rho(x)\} = M_s$ for all $\rho : Var \to M$, iff M_s has only one element.

(3) $M \models \sigma(\varphi_1, \ldots, \varphi_n)$ iff $\overline{\rho}(\sigma(\varphi_1, \ldots, \varphi_n)) = M_s$ for all valuations $\rho : Var \to M$, iff $\sigma_M(\overline{\rho}(\varphi_1), \ldots \overline{\rho}(\varphi_n)) = M_s$ for any $\rho : Var \to M$.
(4) $M \models \neg\varphi$ iff $\overline{\rho}(\neg\varphi) = M_s$ for any $\rho : Var \to M$, iff $M_s \setminus \overline{\rho}(\varphi) = M_s$ for any $\rho : Var \to M$, iff $\overline{\rho}(\varphi) = \emptyset$ for any $\rho : Var \to M$.
(5) $M \models \varphi_1 \wedge \varphi_2$ iff $\overline{\rho}(\varphi_1 \wedge \varphi_2) = M_s$ for any $\rho : Var \to M$, iff $\overline{\rho}(\varphi_1) \cap \overline{\rho}(\varphi_2) = M_s$ for any $\rho : Var \to M$, iff $\overline{\rho}(\varphi_1) = M_s$ and $\overline{\rho}(\varphi_2) = M_s$ for any $\rho : Var \to M$, iff $M \models \varphi_1$ and $M \models \varphi_2$.
(6) $M \models \exists x.\varphi_s$ iff $\overline{\rho}(\exists x.\varphi_s) = M_s$ for any $\rho : Var \to M$, iff $\bigcup\{\overline{\rho'}(\varphi_s) \mid \rho' : Var \to M,\ \rho'\restriction_{Var\setminus\{x\}} = \rho\restriction_{Var\setminus\{x\}}\} = M_s$ for any $\rho : Var \to M$, iff (by the first property in this proposition, since $FV(\varphi_s) \subseteq \{x\}$) $\bigcup\{\overline{\rho'}(\varphi_s) \mid \rho' : Var \to M\} = M_s$ for any $\rho : Var \to M$, iff $\bigcup\{\overline{\rho}(\varphi_s) \mid \rho : Var \to M\} = M_s$. In particular, if $\varphi_s = x$ then $\bigcup\{\overline{\rho}(x) \mid \rho : Var \to M\} = M_s$, so $M \models \exists x.x$.
(7) $M \models \varphi_1 \to \varphi_2$ iff $\overline{\rho}(\varphi_1 \to \varphi_2) = M$ for all $\rho : Var \to M$, iff $\overline{\rho}(\neg(\varphi_1 \wedge \neg\varphi_2)) = M$ for all $\rho : Var \to M$, iff $\overline{\rho}(\varphi_1 \wedge \neg\varphi_2) = \emptyset$ for all $\rho : Var \to M$, iff $\overline{\rho}(\varphi_1) \cap (M \setminus \overline{\rho}(\varphi_2)) = \emptyset$ for all $\rho : Var \to M$, iff $\overline{\rho}(\varphi_1) \subseteq \overline{\rho}(\varphi_2)$ for all $\rho : Var \to M$.
(8) Follows from the previous similar properties for $\wedge$ and $\to$.
(9) $M \models \forall x.\varphi$ iff $\overline{\rho}(\forall x.\varphi) = \bigcap\{\overline{\rho'}(\varphi) \mid \rho' : Var \to M,\ \rho'\restriction_{Var\setminus\{x\}} = \rho\restriction_{Var\setminus\{x\}}\} = M$ for all $\rho : Var \to M$, iff $\overline{\rho'}(\varphi) = M$ for all $\rho, \rho' : Var \to M$ with $\rho'\restriction_{Var\setminus\{x\}} = \rho\restriction_{Var\setminus\{x\}}$, iff $\overline{\rho}(\varphi) = M$ for all $\rho : Var \to M$, iff $M \models \varphi$.

Therefore, all properties hold. □

Since $\exists x.x$ is satisfied by all models (by (6) above), we could have also defined $\top$ as $\exists x.x$ instead of as $x \vee \neg x$. Properties (9) and (2) in Proposition 2.6 imply that the pattern $\forall x.x$ is satisfied precisely by the models whose carrier of the sort of x contains only one element.

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 2.7. *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.*

2.4. **Basic Properties.** A natural question is how to formally reason about patterns. Although they can be inductively built with symbols, like terms are, the following result says that pure predicate logic reasoning is sound for matching logic when we regard patterns as predicates. By *pure* predicate logic we mean predicate logic with just predicate symbols, without constants or function symbols. As shown in Section 11, the Substitution axiom of non-pure predicate logics $(\forall x\,.\,\varphi) \to \varphi[t/x]$ is not sound when t is an arbitrary matching logic pattern (it needs to be modified to only allow patterns which interpret to singletons).

Proposition 2.8. *The following properties hold for patterns of any sort $s \in S$, so the Hilbert-style axioms and proof rules that are sound and complete for pure predicate logic* [39]*, are also sound for matching logic, for any sort (more axioms and proof rules are needed for completeness, as shown in Section 11):*

(1) $\models \varphi$, *where φ is a propositional tautology over patterns of sort s.*

(2) *Modus ponens:* $\models \varphi_1$ *and* $\models \varphi_1 \to \varphi_2$ *imply* $\models \varphi_2$.
(3) $\models (\forall x \,.\, \varphi_1 \to \varphi_2) \to (\varphi_1 \to \forall x \,.\, \varphi_2)$ *when* $x \notin FV(\varphi_1)$.
(4) *Universal generalization:* $\models \varphi$ *implies* $\models \forall x \,.\, \varphi$.
(5) *Substitution:* $\models (\forall x \,.\, \varphi) \to \varphi[y/x]$, *with variable* $y \notin FV(\forall x \,.\, \varphi)$ *of same sort as* x.

Proof. Indeed,

(1) Let ψ be a propositional tautology over propositional variables p_1, ..., p_n, such that φ is obtained from ψ by substituting patterns φ_1, ..., φ_n of sort s for propositional variables p_1, ..., p_n, respectively. Let M be any matching logic model, whose carrier of sort s is M_s, and let ρ be any M-valuation. It is well-known that power-sets are Boolean algebras, in our case $(\mathcal{P}(M_s), \neg, \cap)$ with $\neg$ the complement w.r.t. M_s, and that all Boolean algebras are models of propositional calculus. Therefore, no matter how we interpret the variables p_1, ..., p_n as subsets of M_s, in particular as $\overline{\rho}(\varphi_1)$, ..., $\overline{\rho}(\varphi_n)$, respectively, the interpretation of ψ is the entire set M_s. Hence, $\models \varphi$.
(2) If $\rho : Var \to M$ is a matching logic model valuation such that $\overline{\rho}(\varphi_1) = M_s$ and $\overline{\rho}(\varphi_1) \subseteq \overline{\rho}(\varphi_2)$, then it must be that $\overline{\rho}(\varphi_2) = M_s$.
(3) By (7) in Proposition 2.6, it suffices to show that $\overline{\rho}(\forall x \,.\, \varphi_1 \to \varphi_2) \subseteq \overline{\rho}(\varphi_1 \to \forall x \,.\, \varphi_2)$ for any valuation $\rho : Var \to M$. A stronger result (equality) holds, as expected:

$$\begin{array}{rcl}
\overline{\rho}(\forall x \,.\, \varphi_1 \to \varphi_2) & = & \bigcap\{\overline{\rho'}(\varphi_1 \to \varphi_2) \mid \rho' : Var \to M,\ \rho'{\restriction}_{Var\setminus\{x\}} = \rho{\restriction}_{Var\setminus\{x\}}\} \\
& = & \bigcap\{M_s \setminus (\overline{\rho'}(\varphi_1) \setminus \overline{\rho'}(\varphi_2)) \mid \rho' : Var \to M,\ \rho'{\restriction}_{Var\setminus\{x\}} = \rho{\restriction}_{Var\setminus\{x\}}\} \\
& = & \bigcap\{M_s \setminus (\overline{\rho}(\varphi_1) \setminus \overline{\rho'}(\varphi_2)) \mid \rho' : Var \to M,\ \rho'{\restriction}_{Var\setminus\{x\}} = \rho{\restriction}_{Var\setminus\{x\}}\} \\
& & \text{(by (1) in Proposition 2.6, because } x \notin FV(\varphi_1)) \\
& = & M_s \setminus (\overline{\rho}(\varphi_1) \setminus \bigcap\{\overline{\rho'}(\varphi_2)) \mid \rho' : Var \to M,\ \rho'{\restriction}_{Var\setminus\{x\}} = \rho{\restriction}_{Var\setminus\{x\}}\} \\
& & \text{(set theory properties of relative complements)} \\
& = & M_s \setminus (\overline{\rho}(\varphi_1) \setminus \overline{\rho}(\forall x \,.\, \varphi_2) \\
& = & \overline{\rho}(\varphi_1 \to \forall x \,.\, \varphi_2)
\end{array}$$

(4) Immediate by (9) in Proposition 2.6.
(5) Follows by (7) and (1) in Proposition 2.6, because for any valuation $\rho : Var \to M$, we have $\overline{\rho}(\varphi[y/x]) = \overline{\rho'}(\varphi)$ where $\rho' : Var \to M$ is defined as $\rho'{\restriction}_{Var\setminus\{x\}} = \rho{\restriction}_{Var\setminus\{x\}}$ and $\rho'(x) = \rho(y)$ (this holds also when $y = x$), while $\overline{\rho}(\forall x \,.\, \varphi)$ is the intersection of all $\overline{\rho''}(\varphi)$ for all $\rho'' : Var \to M$ with $\rho''{\restriction}_{Var\setminus\{x\}} = \rho{\restriction}_{Var\setminus\{x\}}$ and ρ' is one of these ρ''.

Therefore, pure predicate logic reasoning can also be used to reason about patterns. □

Proposition 2.8 tells us that the proof system of pure predicate logic is actually sound for matching logic, *unchanged*. That is, we do not need to attempt to translate patterns to predicate logic formulae in order to reason about them, we can simply regard them as predicates the way they are. Section 11 shows that a few additional proof rules yield a sound and complete proof system for matching logic, similarly to how (term) Substitution together with the other four proof rules of pure predicate logic brings complete deduction to FOL.

Sometimes we can show that patterns are two-valued:

Definition 2.9. *Pattern* φ *is called a* ***predicate*** *in* (S, Σ, F), *or simply a* ***predicate*** *when* (S, Σ, F) *is understood, iff it is an* M*-predicate (Definition 2.4) in all models* M *with* $M \models F$.

However, note that Proposition 2.8 applies to any patterns, not only to predicates. Moreover, there are also interesting properties that appear to be very specific to patterns and their dual logical-structural nature, and not to predicates, such as the following:

Proposition 2.10. ***(Structural Framing)*** *If $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ and $\varphi_i, \varphi_i' \in \text{PATTERN}_{s_i}$ such that $\models \varphi_i \to \varphi_i'$ for all $i \in 1 \ldots n$, then $\models \sigma(\varphi_1, \ldots, \varphi_n) \to \sigma(\varphi_1', \ldots, \varphi_n')$.*

Proof. Immediate by (7) in Proposition 2.6, because for any model M, the extension of σ_M as a function $\mathcal{P}(M_{s_1}) \times \cdots \times \mathcal{P}(M_{s_n}) \to \mathcal{P}(M_s)$ is monotone. □

This structural framing property generalizes to *positive*, or *monotone* contexts: if $\models \varphi \to \varphi'$ then $\models C[\varphi] \to C[\varphi']$ for any positive context C. By a positive/monotone context we mean a context with no negation on the path to the placeholder.[1] Indeed, except for $\neg$, the matching logic constructs are interpreted as monotone functions over powerset domains. Structural framing is crucial for localizing reasoning. Consider, for example, the property

$$\models \ (1 \mapsto 5 * 2 \mapsto 0 * 7 \mapsto 9 * 8 \mapsto 1) \to \mathit{list}(7, 9 \cdot 5)$$

proved in Section 11 for the matching logic specifications of maps (which captures separation logic: Section 9). Taking σ as the map/heap merge operation $\_ * \_$, Proposition 2.10 implies

$$\models \ (1 \mapsto 5 * 2 \mapsto 0 * 7 \mapsto 9 * 8 \mapsto 1 * h) \to \mathit{list}(7, 9 \cdot 5) * h$$

where h is a free map/heap variable. So the property we "locally" proved can be "framed" within any map/heap. Of course, one can go further and "globalize" the property in any positive context. For example, consider the operational semantics of a real language like C, whose configuration was partly discussed in the example in Section 2.2. Recall from Section 2.2 that semantic cells, written using symbols $\langle ... \rangle_{\mathsf{cell}}$, can be nested and their grouping (symbol) is governed by associativity and commutativity axioms. Also, there is a top cell $\langle ... \rangle_{\mathsf{cfg}}$ holding a subcell $\langle ... \rangle_{\mathsf{heap}}$ among many others. Proposition 2.6 then implies

$$\models \ \ \langle \langle 1 \mapsto 5 * 2 \mapsto 0 * 7 \mapsto 9 * 8 \mapsto 1 * h \rangle_{\mathsf{heap}} \ c \rangle_{\mathsf{cfg}} \ \to \ \langle \langle \mathit{list}(7, 9 \cdot 5) * h \rangle_{\mathsf{heap}} \ c \rangle_{\mathsf{cfg}}$$

where h and c are free variables (the "heap" and, respectively, "configuration" frames).

As discussed in the example in Section 2.2, sometimes it is useful to move the logical connectives from inside terms to the top level, or viceversa. While disjunction and existential quantification can be propagated both ways through symbol applications ($\leftrightarrow$), conjunction and universal quantification weaken the pattern as they are propagated from the inside to the outside of a symbol application ($\to$), and negation appears to not be movable at all:

Proposition 2.11. ***(Distributivity of symbol application)*** *Let $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ and $\varphi_i \in \text{PATTERN}_{s_i}$ for all $1 \leq i \leq n$. Pick a particular $1 \leq i \leq n$. Let $\varphi_i' \in \text{PATTERN}_{s_i}$ be another pattern of sort s_i and let $C_{\sigma,i}[\Box]$ be the context $\sigma(\varphi_1, \ldots, \varphi_{i-1}, \Box, \varphi_{i+1}, \ldots \varphi_n)$ (a context $C[\Box]$ is a pattern with one occurrence of a free variable, "$\Box$", and $C[\varphi]$ is $C[\varphi/\Box]$). Then:*

(1) $\models C_{\sigma,i}[\varphi_i \vee \varphi_i'] \leftrightarrow C_{\sigma,i}[\varphi_i] \vee C_{\sigma,i}[\varphi_i']$
(2) $\models C_{\sigma,i}[\exists x \,.\, \varphi_i] \leftrightarrow \exists x \,.\, C_{\sigma,i}[\varphi_i]$, *where* $x \notin FV(C_{\sigma,i}[\Box])$
(3) $\models C_{\sigma,i}[\varphi_i \wedge \varphi_i'] \to C_{\sigma,i}[\varphi_i] \wedge C_{\sigma,i}[\varphi_i']$
(4) $\models C_{\sigma,i}[\forall x \,.\, \varphi_i] \to \forall x \,.\, C_{\sigma,i}[\varphi_i]$, *where* $x \notin FV(C_{\sigma,i}[\Box])$

Proof. Trivial, using the basic set properties that for any function $f : X \to \mathcal{P}(Y)$ (i.e., relation in $X \times Y$), if $\{A_i\}_{i \in \mathcal{I}}$ is a family of subsets of X, i.e., $A_i \subseteq X$ for all $i \in \mathcal{I}$, then $f(\bigcup\{A_i \mid i \in \mathcal{I}\}) = \bigcup\{f(A_i) \mid i \in \mathcal{I}\}$ and $f(\bigcap\{A_i \mid i \in \mathcal{I}\}) \subseteq \bigcap\{f(A_i) \mid i \in \mathcal{I}\}$, where $f(A) = \bigcup\{f(a) \mid a \in A\}$. Note the inclusion for intersection, as opposed to equality for

[1] In the context of programming language semantics, reasoning typically happens in semantic cells in the program configuration and the program configuration is typically a term with variables, possibly domain-constrained, so requiring a context to be positive is not a strong requirement.

disjunction. The inclusion for intersection becomes equality when f is injective as a relation, that is, when $f(a) \cap f(a') \neq \emptyset$ implies $a = a'$. □

The other implications in (3) and (4) above in Proposition 2.11 do not hold in general. Consider a signature Σ containing only one sort, two constants a and b, and a binary symbol f. Consider also a model M containing only two elements, a_M and b_M, with constants a and b interpreted as $\{a_M\}$ and $\{b_M\}$, respectively, and with f interpreted as the injective function $f_M(a_M, a_M) = \{a_M\}$, $f_M(b_M, a_M) = \{b_M\}$, $f_M(a_M, b_M) = \{b_M\}$, $f_M(b_M, b_M) = \{a_M\}$. Let $C_{f,2}[\Box]$ be the context $f(a \vee b, \Box)$ and let φ_2 and φ_2' be a and b, respectively. Then the pattern $C_{f,2}[a] \wedge C_{f,2}[b]$, that is $f(a \vee b, a) \wedge f(a \vee b, b)$, is interpreted by any valuation to M as the (total) set $\{a_M, b_M\}$, while $C_{f,2}[a \wedge b]$, that is $f(a \vee b, a \wedge b)$, as the empty set (because $a \wedge b$ is interpreted as the empty set). Therefore, $\not\models C_{f,2}[a] \wedge C_{f,2}[b] \to C_{f,2}[a \wedge b]$. Similarly, $\not\models \forall x \,.\, C_{f,2}[x] \to C_{f,2}[\forall x \,.\, x]$ because $\forall x \,.\, C_{f,2}[x]$ and $C_{f,2}[\forall x \,.\, x]$ are interpreted as $\{a_M, b_M\}$ and $\emptyset$, respectively, by any valuation to M.

The reason for which the counter-examples above worked was that the context $C_{f,2}[\Box]$, that is $f(a \vee b, \Box)$, did not yield an injective relation in M: indeed, it was not the case that the interpretations of $f(a \vee b, x)$ and $f(a \vee b, y)$ were disjoint whenever x and y were interpreted as distinct elements. We can define a general notion of injectivity, for any context $C_{\sigma,i}[\Box]$, which generalizes the usual notion of injectivity of a function or relation:

Definition 2.12. *With the notation in Proposition 2.11, $C_{\sigma,i}[\Box]$ is **injective** in specification (S, Σ, F) iff $F \models C_{\sigma,i}[x] \wedge C_{\sigma,i}[y] \to C_{\sigma,i}[x \wedge y]$, where $x, y \in Var_{s_i}$ are distinct variables which do not occur in $C_{\sigma,i}[\Box]$. We drop (S, Σ, F) when understood. Symbol σ is **injective on position** i iff $C_{\sigma,i}[\Box]$ is injective with φ_1, ..., φ_{i-1}, φ_{i+1},...,φ_n chosen as distinct variables.*

It is easy to check that σ is injective on position i iff for any model M with $M \models F$, σ_M is injective on position i as a relation in M. Recall that functions are particular relations, and that injectivity is a property of relations in general: $R \subseteq M_{s_1} \times M_{s_n} \times M_s$ is injective on position $1 \leq i \leq$ iff $(a_1, ..., a_{i-1}, a_i, a_{i+1}, ..., a_n, b) \in R$ and $(a_1, ..., a_{i-1}, a_i', a_{i+1}, ..., a_n, b) \in R$ implies $a_i = a_i'$. Regarding σ_M as such a relation, its injectivity on position i means that $\sigma_M(a_1, ..., a_{i-1}, a_i, a_{i+1}, ..., a_n) \cap \sigma_M(a_1, ..., a_{i-1}, a_i', a_{i+1}, ..., a_n) \neq \emptyset$ implies $a_i = a_i'$.

Proposition 2.13. ***(Distributivity of injective symbol application)*** *With the notation in Definition 2.12, if $C_{\sigma,i}[\Box]$ is injective in (S, Σ, F) and $\varphi_i, \varphi_i' \in \text{PATTERN}_{s_i}$ then:*

(1) $F \models C_{\sigma,i}[\varphi_i] \wedge C_{\sigma,i}[\varphi_i'] \to C_{\sigma,i}[\varphi_i \wedge \varphi_i']$
(2) $F \models \forall x \,.\, C_{\sigma,i}[\varphi_i] \to C_{\sigma,i}[\forall x \,.\, \varphi_i]$, *where* $x \notin FV(C_{\sigma,i}[\Box])$

Together with Proposition 2.11, this implies the full distributivity of injective contexts w.r.t. the matching logic constructs $\wedge$, $\vee$, $\forall$, $\exists$ (but not $\neg$).

Proof. Let M be a model with $M \models F$ and let $\rho : Var \to M$ be a valuation.

To prove the first property, let $b \in \overline{\rho}(C_{\sigma,i}[\varphi_i] \wedge C_{\sigma,i}[\varphi_i'])$, that is, $b \in \overline{\rho}(C_{\sigma,i}[\varphi_i])$ and $b \in \overline{\rho}(C_{\sigma,i}[\varphi_i'])$. Then there are $a, a' \in M_{s_i}$ such that $a \in \overline{\rho}(\varphi_i)$ and $b \in \overline{\rho[a/\Box]}(C_{\sigma,i}[\Box])$, and $a' \in \overline{\rho}(\varphi_i')$ and $b \in \overline{\rho[a'/\Box]}(C_{\sigma,i}[\Box])$. Let $x, y \in Var_{s_i}$ be two distinct variables that do not occur in $C_{\sigma,i}[\Box]$ and let ρ' be the valuation $\rho[a/x][a'/y]$. Then we have $b \in \overline{\rho'}(C_{\sigma,i}[x])$ and $b \in \overline{\rho'}(C_{\sigma,i}[y])$, that is, $b \in \overline{\rho'}(C_{\sigma,i}[x] \wedge C_{\sigma,i}[y])$. The injectivity hypothesis then implies $b \in \overline{\rho'}(C_{\sigma,i}[x \wedge y])$. Therefore, $\overline{\rho'}(x \wedge y)$ is non-empty, that is, $\{a\} \cap \{a'\}$ is non-empty, that is, $a = a'$. Since $a \in \overline{\rho}(\varphi_i)$ and $a' \in \overline{\rho}(\varphi_i')$, it follows that $a \in \overline{\rho}(\varphi_i \wedge \varphi_i')$. Since $b \in \overline{\rho[a/\Box]}(C_{\sigma,i}[\Box])$, it follows that $b \in \overline{\rho}(C_{\sigma,i}[\varphi_i \wedge \varphi_i'])$.

For the second, let $b \in \overline{\rho}(\forall x \,.\, C_{\sigma,i}[\varphi_i])$, that is, $b \in \overline{\rho[v/x]}(C_{\sigma,i}[\varphi_i])$ for all $v \in M_{sort(x)}$, that is, for any $v \in M_{sort(x)}$ there is some $a_v \in \overline{\rho[v/x]}(\varphi_i)$ such that $b \in \overline{\rho[a_v/\Box]}(C_{\sigma,i}[\Box])$ (because $x \notin FV(C_{\sigma,i}[\Box])$, so $\overline{\rho[v/x][a_v/\Box]}(C_{\sigma,i}[\Box]) = \overline{\rho[a_v/\Box]}(C_{\sigma,i}[\Box])$). The injectivity of $C_{\sigma,i}[\Box]$ implies that all such a_v elements are equal. Indeed, let $v, v' \in M_{sort(x)}$ and $a_v \in \overline{\rho[v/x]}(\varphi_i)$ and $a_{v'} \in \overline{\rho[v'/x]}(\varphi_i)$ such that $b \in \overline{\rho[a_v/\Box]}(C_{\sigma,i}[\Box])$ and $b \in \overline{\rho[a_{v'}/\Box]}(C_{\sigma,i}[\Box])$. Let $z, y \in Var_{s_i}$ be two distinct variables that do not occur in $C_{\sigma,i}[\Box]$, like in the Definition 2.12 of injectivity (but with z instead of x to avoid name collision), and note that the above implies $b \in \overline{\rho[a_v/z][a_{v'}/y]}(C_{\sigma,i}[z] \wedge C_{\sigma,i}[y])$. Then Definition 2.12 implies $b \in \overline{\rho[a_v/z][a_{v'}/y]}(C_{\sigma,i}[z \wedge y])$. Therefore $\overline{\rho[a_v/z][a_{v'}/y]}(z \wedge y) \neq \emptyset$, that is, $a_v = a_{v'}$. Since all the elements $a_v \in \overline{\rho[v/x]}(\varphi_i)$ for all $v \in M_{sort(x)}$ are equal, it follows that there is some element $a \in \overline{\rho}(\forall x \,.\, \varphi_i)$ such that $a_v = a$ for all a_v as above. Moreover, $b \in \overline{\rho[a/\Box]}(C_{\sigma,i}[\Box])$, that is, $b \in \overline{\rho}(C_{\sigma,i}[\forall x \,.\, \varphi_i])$. □

The notion of context injectivity in Definition 2.12 is the weakest theoretical condition we were able to find in order for the (bidirectional) distributivity of conjunction and universal quantification to hold. In practice, stronger conditions are met. For example, Section 5.7 discusses constructors, which are symbols whose interpretations are injective in all their arguments at the same time (i.e., $\sigma_M(a_1, ..., a_n) \cap \sigma_M(a'_1, ..., a'_n) \neq \emptyset$ implies $a_1 = a'_1$, ..., $a_n = a'_n$). Contexts corresponding to constructors are injective in the sense of Definition 2.12.

We next demonstrate the usefulness of matching logic by a series of other examples.

3. Instance: Propositional Calculus

In Section 2, (1) in Proposition 2.8, we showed that propositional reasoning is sound for matching logic. Here we go one step further and show that we can can instantiate matching logic to become precisely propositional calculus, without any translation needed in any direction. The idea is to add a special sort for propositions, say *Prop*, then to use the already existing syntax of matching logic to build propositions as we know them, and then to show that the existing semantics of matching logic, given by $\models$, yields the expected semantics of propositions as we know it in propositional calculus (let us refer to it as $\models_{Prop}$).

We build a matching logic signature as follows: S contains only one sort, *Prop*, and Σ is empty. Let us also drop the existential quantifier, so that the resulting syntax of patterns becomes exactly that of propositional calculus:

$$\begin{array}{rcl} \varphi & ::= & Var_{Prop} \\ & | & \neg\varphi \\ & | & \varphi \wedge \varphi \end{array}$$

Then the default matching logic semantics endows the resulting syntax of propositions with the desired propositional calculus semantics:

Proposition 3.1. *For any proposition φ, the following holds: $\models_{Prop} \varphi$ iff $\models \varphi$.*

Proof. The implication "$\models_{Prop} \varphi$ implies $\models \varphi$" follows by (1) in Proposition 2.8. For the other implication, let us suppose that $\models \varphi$ and let $\theta : Var_{Prop} \to \{true, false\}$ be an arbitrary propositional valuation (it is often called a "model" in the literature, but we refrain from using that terminology to avoid confusion with our notion of model). All we have to do is show that $\theta(\varphi) = true$. Let M be the matching logic model with $M_{Prop} = \{true, false\}$ and let $\rho_\theta : Var \to M$ be the matching logic valuation where $\rho_\theta(x) = \theta(x)$ for each $x \in Var_{Prop}$.

Note that, unlike in propositional calculus where propositions ψ evaluate to precisely one of *true* or *false* for any given valuation θ, in matching logic $\overline{\rho_\theta}(\psi)$ can be any of the four subsets of $\{true, false\}$. For example, if x and y are variables such that $\theta(x) = true$ and $\theta(y) = false$, then $\overline{\rho_\theta}(x) = \{true\}$, $\overline{\rho_\theta}(y) = \{false\}$, $\overline{\rho_\theta}(\neg x) = \{false\}$, $\overline{\rho_\theta}(\neg y) = \{true\}$, $\overline{\rho_\theta}(x \wedge y) = \emptyset$, $\overline{\rho_\theta}(x \vee y) = \{true, false\}$. Nevertheless, we can inductively show that the propositional validity of a proposition ψ is dictated by the membership of *true* to its matching logic evaluation as a set: $\theta(\psi) = true$ iff $true \in \overline{\rho_\theta}(\psi)$. Indeed: if ψ is a variable x then $\overline{\rho_\theta}(x) = \{\theta(x)\}$, so the property holds; if ψ is $\neg\psi'$ then $\overline{\rho_\theta}(\neg\psi') = \{true, false\} \backslash \overline{\rho_\theta}(\psi')$, so $true \in \overline{\rho_\theta}(\neg\psi')$ iff $true \notin \overline{\rho_\theta}(\psi')$, iff (by the induction hypothesis) $\theta(\psi') \neq true$, iff (by the two-valued semantics of propositional calculus) $\theta(\neg\psi') = true$; finally, if ψ is $\psi_1 \wedge \psi_2$ then $true \in \overline{\rho_\theta}(\psi_1 \wedge \psi_2)$ iff $true \in \overline{\rho_\theta}(\psi_1)$ and $true \in \overline{\rho_\theta}(\psi_2)$, iff (by the induction hypothesis) $\theta(\psi_1) = true$ and $\theta(\psi_2) = true$, iff $\theta(\psi_1 \wedge \psi_2) = true$.

Now $\models \varphi$ implies $\overline{\rho_\theta}(\varphi) = \{true, false\}$, so $true \in \overline{\rho_\theta}(\varphi)$. By the result proved inductively above we conclude that $\theta(\varphi) = true$. □

An alternative way to capture propositional logic is to add a constant symbol (i.e., a symbol without any arguments) to Σ for each propositional variable, like we do for modal logic in Section 8. This is similar to how predicate logic captures propositional calculus, namely by associating a predicate without arguments to each propositional variable. We leave the details as an exercise to the interested reader.

4. Instance: (Pure) Predicate Logic

Recall from Section 2, Proposition 2.8 and the discussion preceding it, that by *pure* predicate logic in this paper we mean predicate logic or first-order logic (FOL) with only predicate symbols (no function and no constant symbols). Note that some works call the fragment of FOL with only constant (i.e., zero-argument function) symbols "predicate logic", others use "predicate logic" as a synonym for FOL. We do not discuss the fragment of FOL with only constant symbols in this paper, so from here on we take the liberty to refer to "pure predicate logic" as just "predicate logic". Proposition 2.8 showed that predicate logic reasoning is sound for matching logic. Similarly to propositional calculus in Section 3, here we go one step further and show that we can can instantiate matching logic to become precisely predicate logic; the FOL case will be discussed in Section 7. We follow the same approach like for propositional calculus: add a special sort for predicates, say *Pred*, then use the already existing syntax of matching logic to build formulae as we know them in predicate logic, and then show that the existing semantics of matching logic, given by $\models$, yields the expected semantics of pure predicate logic. We let $\models_{PL}$ denote the predicate logic satisfaction.

Recall that predicate logic is the fragment of first-order logic with just predicate symbols, that is, with no function (including no constant) and no equality symbols. We consider only the many-sorted case here. Formally, if S is a sort set and Π is a set of predicate symbols, the syntax of pure predicate logic formulae is

$$\begin{array}{rcl} \varphi & ::= & \pi(x_1, \ldots, x_n) \text{ with } \pi \in \Pi_{s_1 \ldots s_n},\ x_1 \in Var_{s_1},\ \ldots,\ x_n \in Var_{s_n} \\ & | & \neg\varphi \\ & | & \varphi \wedge \varphi \\ & | & \exists x \,.\, \varphi \end{array}$$

Without loss of generality, suppose that we can pick a fresh sort name, *Pred*; that is, $Pred \notin S$. Let us now construct the matching logic signature $(S \cup \{Pred\}, \Sigma)$, where

$\Sigma_{s_1 \ldots s_n, Pred} = \Pi_{s_1 \ldots s_n}$ are the only symbols in Σ; that is, Σ contains precisely the predicate symbols of the predicate logic signature, but regarded as pattern symbols of result sort *Pred*. Suppose also that we disallow any variables of sort *Pred* in patterns. Then the matching logic patterns of sort *Pred* are precisely the predicate logic formulae, without any translation in any direction. Moreover, the following result shows that the default matching logic semantics endows these patterns with their desired predicate logic semantics:

Proposition 4.1. *For any predicate logic formula φ, the following holds: $\models_{PL} \varphi$ iff $\models \varphi$.*

Proof. That $\models_{PL} \varphi$ implies $\models \varphi$ follows by Proposition 2.8: each of the proof rules of the complete proof system of (pure) predicate logic [39] is sound for matching logic. For the other implication, note that we can associate to any predicate logic model $M^{PL} = (\{M^{PL}_s\}_{s \in S}, \{\pi_{M^{PL}}\}_{\pi \in \Pi})$ a matching logic model $M^{ML} = (\{M^{ML}_s\}_{s \in S \cup \{Pred\}}, \{\pi_{M^{ML}}\}_{\pi \in \Sigma})$, where $M^{ML}_s = M^{PL}_s$ for all $s \in S$ and $M^{ML}_{Pred} = \{\star\}$ (with $\star$ some arbitrary but fixed element) and $\pi_{M^{ML}}(a_1, \ldots, a_n) = \{\star\}$ iff $\pi_{M^{PL}}(a_1, \ldots, a_n)$ holds, and $\pi_{M^{ML}}(a_1, \ldots, a_n) = \emptyset$ otherwise. Furthermore, we can show that for any PL formula φ, we have $M^{PL} \models_{PL} \varphi$ iff $M^{ML} \models_{ML} \varphi$. Since φ does not contain any variables of sort *Pred*, by (1) in Proposition 2.6 it suffices to show that for any $\rho : Var \to M^{PL}$, it is the case that $M^{PL}, \rho \models_{PL} \varphi$ iff $\overline{\rho}(\varphi) = \{\star\}$. We can easily show this property by structural induction on φ. The only relatively non-trivial case is the complement construct, which shows why it was important for M^{ML}_{Pred} to contain precisely one element: $M^{PL}, \rho \models_{PL} \neg\varphi$ iff $M^{PL}, \rho \not\models_{PL} \varphi$ iff (by the induction hypothesis) $\overline{\rho}(\varphi) \neq \{\star\}$ iff $\overline{\rho}(\varphi) = \emptyset$ iff $\overline{\rho}(\neg\varphi) = \{\star\}$.

Therefore, $M^{PL} \models_{PL} \varphi$ iff $M^{ML} \models_{ML} \varphi$. Since the predicate logic model M^{PL} was chosen arbitrarily, it follows that $\models \varphi$ implies $\models_{PL} \varphi$. □

5. Matching Logic: Useful Symbols and Notations

Here we show how to define, in matching logic, several mathematical instruments of practical importance, such as equality, membership, and functions. We also introduce appropriate notations for them, because they will be used frequently and tacitly in the rest of the paper.

The role of this section is twofold. On the one hand, it illustrates the expressiveness of matching logic. Indeed, we can define all the crucial mathematical notions above as matching logic specifications or as syntactic sugar, without any changes to the matching logic itself (recall, for example, that equality cannot be defined in first-order logics; the logic itself needs to be modified into "first-order logic with equality"—more details in Section 5.2). On the other hand, it shows that despite the apparently non-conventional interpretation of patterns as sets of values in matching logic, the conventional mathematical machinery used to reason about program states is still available, with its expected meaning.

Unless otherwise mentioned, for the rest of this section we assume an arbitrary but fixed matching logic specification (S, Σ, F).

5.1. Definedness and Totality.

In classical logics, the interpretation of a formula under a given valuation is either true or false, and there is only one syntactic category for formulae while multiple syntactic categories for data. In contrast, matching logic patterns are interpreted as sets of values, those that match them, where the total set corresponds to the intution of "true", or $\top$, and the empty set corresponds to "false", or $\bot$. Also, each matching logic syntactic category, or sort, admits both data constructs and its own logical connectives and quantifiers. These leave two questions open:

(1) How can we interpret patterns in a conventional, two-valued way? Are the patterns matched by proper (i.e., neither total nor empty) subsets of elements true, or false?
(2) How can we lift reasoning within syntactic category s_1 to syntactic category s_2?

These questions are particularly important when attempting to combine matching logic reasoning with classical reasoning or provers for existing mathematical domains.

It turns out that the above can be methodologically achieved by adding some symbols and defining patterns for them to the matching logic specification (S, Σ, F). Specifically, for any pair of sorts of interest $s_1, s_2 \in S$, which need not be distinct, we can add a symbol $\lceil \_ \rceil^{s_2}_{s_1}$ to Σ_{s_1,s_2} and an axiom pattern to F that makes $\lceil \_ \rceil^{s_2}_{s_1}$ behave like a *definedness predicate* for any pattern of sort s_1, with two-valued result of sort s_2: $\lceil \varphi \rceil^{s_2}_{s_1}$ is either $\bot_{s_2}$ when φ is $\bot_{s_1}$, or $\top_{s_2}$ otherwise (i.e., if φ is matched by some values of sort s_1). The pattern that we can add to F in order to achieve the above is in fact unexpectedly simple: $\lceil x{:}s_1 \rceil^{s_2}_{s_1}$.

Although we do not need it for many of the subsequent results, to simplify the overall presentation of the rest of the paper, from here on we tacitly work under the following:

Assumption 5.1. *For any (not necessarily distinct) sorts $s_1, s_2 \in S$, assume the following:*

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

*We call the symbols $\lceil \_ \rceil^{s_2}_{s_1}$ **definedness symbols**.*

We next show that the definedness symbol indeed has the expected meaning:

Proposition 5.2. *If $\varphi \in \text{PATTERN}_{s_1}$ then $\lceil \varphi \rceil^{s_2}_{s_1}$ is a predicate (Definition 2.9). Specifically, if $\rho : Var \to M$ is any valuation 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).*

Proof. By Definition 2.3, $\overline{\rho}(\lceil \varphi \rceil^{s_2}_{s_1}) = (\lceil \_ \rceil^{s_2}_{s_1})_M(\overline{\rho}(\varphi))$. The definedness pattern axiom states that $\lceil x : s_1 \rceil^{s_2}_{s_1}$ is valid (Assumption 5.1), which implies $(\lceil \_ \rceil^{s_2}_{s_1})_M(m_1) = M_{s_2}$ for all $m_1 \in M_{s_1}$, so if there is any $m_1 \in \overline{\rho}(\varphi)$ then $(\lceil \_ \rceil^{s_2}_{s_1})_M(\overline{\rho}(\varphi))$ can only be M_{s_2}. On the other hand, if $\overline{\rho}(\varphi) = \emptyset$ then $(\lceil \_ \rceil^{s_2}_{s_1})_M(\overline{\rho}(\varphi)) = \emptyset$. □

Notation 5.3. *We also define **totality**, $\lfloor \_ \rfloor^{s_2}_{s_1}$, as a derived construct dual to definedness:*

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

The totality construct states that the enclosed pattern must be matched by all values:

Proposition 5.4. *If $\varphi \in \text{PATTERN}_{s_1}$ then $\lfloor \varphi \rfloor^{s_2}_{s_1}$ is a predicate (Definition 2.9). Specifically, 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 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 total).*

Proof. $\overline{\rho}(\lfloor \varphi \rfloor^{s_2}_{s_1}) = \overline{\rho}(\neg \lceil \neg \varphi \rceil^{s_2}_{s_1}) = M_{s_2} \backslash \overline{\rho}(\lceil \neg \varphi \rceil^{s_2}_{s_1})$. So $\overline{\rho}(\lfloor \varphi \rfloor^{s_2}_{s_1}) = \emptyset$ iff $\overline{\rho}(\lceil \neg \varphi \rceil^{s_2}_{s_1}) = M_{s_2}$ iff $\overline{\rho}(\neg \varphi) \neq \emptyset$ (by Proposition 5.2) iff $\overline{\rho}(\varphi) \neq M_{s_1}$. Similarly, $\overline{\rho}(\lfloor \varphi \rfloor^{s_2}_{s_1}) = M_{s_2}$ iff $\overline{\rho}(\lceil \neg \varphi \rceil^{s_2}_{s_1}) = \emptyset$ iff $\overline{\rho}(\neg \varphi) = \emptyset$ (by Proposition 5.2) iff $\overline{\rho}(\varphi) = M_{s_1}$. □

Totality is useful, for example, to define pattern equality as the totality of the pattern equivalence relation; this is discussed in depth shortly (Section 5.2). It is also useful when there is a need to restrict a pattern context, say φ of sort s_2, to only instances where pattern φ_1 of sort s_1 implies pattern φ_2 of sort s_1: $\varphi \wedge \lfloor \varphi_1 \to \varphi_2 \rfloor^{s_2}_{s_1}$. Indeed, $\overline{\rho}(\varphi \wedge \lfloor \varphi_1 \to \varphi_2 \rfloor^{s_2}_{s_1})$ is $\overline{\rho}(\varphi)$ iff $\overline{\rho}(\varphi_1) \subseteq \overline{\rho}(\varphi_2)$, and it is $\emptyset$ otherwise. For example, $\exists x \,.\, x \wedge \lfloor \varphi_1 \to \varphi_2 \rfloor^{s_2}_{s_1}$ defines the set of all values of x with the property that if they match φ_1 then they also match φ_2. A concrete instance of this is the definition of "magic wand" in separation logic (Section 9).

The totality constructs satisfy, in a more general sorted setting, some of the basic properties of modal logic operators, such as **(N)**, **(K)**, **(M)** and **(5)** [7, 57, 45]:

Corollary 5.5. *If $s_1, s_2 \in S$ and φ, φ_1 and φ_2 are patterns of sort s_1, then:*

(N) *If* $\models \varphi$ *then* $\models \lfloor \varphi \rfloor^{s_2}_{s_1}$
(K) $\models \lfloor \varphi_1 \to \varphi_2 \rfloor^{s_2}_{s_1} \to (\lfloor \varphi_1 \rfloor^{s_2}_{s_1} \to \lfloor \varphi_2 \rfloor^{s_2}_{s_1})$
(M) $\models \lfloor \varphi \rfloor^{s_1}_{s_1} \to \varphi$
(5) $\models \lceil \varphi \rceil^{s_2}_{s_1} \to \lfloor \lceil \varphi \rceil^{s_2}_{s_1} \rfloor^{s_2}_{s_2}$

Proof. The **(N)** property is an immediate corollary of Proposition 5.4. For the **(K)** property, let M be a model and $\rho : Var \to M$ a valuation. By Proposition 5.4 and the discussion in the paragraph following it, $\overline{\rho}(\lfloor \varphi_1 \to \varphi_2 \rfloor^{s_2}_{s_1})$ is either $\emptyset$ or M_{s_2}, the latter happening iff $\overline{\rho}(\varphi_1) \subseteq \overline{\rho}(\varphi_2)$. The first case makes our property vacuously hold. In the second case, we have to show that $\overline{\rho}(\lfloor \varphi_1 \rfloor^{s_2}_{s_1} \to \lfloor \varphi_2 \rfloor^{s_2}_{s_1}) = M_{s_2}$, that is, that $\overline{\rho}(\lfloor \varphi_1 \rfloor^{s_2}_{s_1}) \subseteq \overline{\rho}(\lfloor \varphi_2 \rfloor^{s_2}_{s_1})$, which follows by Proposition 5.4 from $\overline{\rho}(\varphi_1) \subseteq \overline{\rho}(\varphi_2)$. To show **(M)**, we have to show $\overline{\rho}(\lfloor \varphi \rfloor^{s_1}_{s_1}) \subseteq \overline{\rho}(\varphi)$ for any $\rho : Var \to M$. By Proposition 5.4, we only need to consider the case where $\overline{\rho}(\lfloor \varphi \rfloor^{s_1}_{s_1}) = M_{s_1}$; but this can only happen when $\overline{\rho}(\varphi) = M_{s_1}$, so the property holds. For **(5)**, let $\rho : Var \to M$ be such that $\overline{\rho}(\lceil \varphi \rceil^{s_2}_{s_1}) = M_{s_2}$ (by Proposition 5.2, the only other case is $\overline{\rho}(\lceil \varphi \rceil^{s_2}_{s_1}) = \emptyset$, so the property holds vacuously for that case). Then by Proposition 5.4 it follows that $\overline{\rho}(\lfloor \lceil \varphi \rceil^{s_2}_{s_1} \rfloor^{s_2}_{s_2}) = M_{s_2}$, so $\overline{\rho}(\lceil \varphi \rceil^{s_2}_{s_1}) \subseteq \overline{\rho}(\lfloor \lceil \varphi \rceil^{s_2}_{s_1} \rfloor^{s_2}_{s_2})$. □

In Section 8 we show that the modal logic S5 is equivalent to a matching logic specification, where the definedness and totality constructs play the role of the $\Diamond$ and $\Box$ modalities.

Notation 5.6. *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. 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 $x \in Var$" 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.

Notation 5.7. *If φ is a predicate (Definition 2.9, then we write $[\varphi]$ instead of $\lceil \varphi \rceil$ or $\lfloor \varphi \rfloor$. This notation is justified, because if φ is a predicate then $\models \lceil \varphi \rceil \leftrightarrow \lfloor \varphi \rfloor$.*

As Proposition 5.12 will shortly show, if φ is a predicate, then by "wrapping" it with square brackets, as $[\varphi]$, we can propagate it through the configuration symbols and conjunctive constraints to wherever it is needed, to facilitate local reasoning.

5.2. **Equality.** Here we show that, unlike in predicate logic or FOL, equality can be defined in matching logic. Before that, let us recall why *equality cannot be defined in FOL*. We only give a short intuitive explanation here; the interested reader is referred to authoritative FOL textbooks for full details, e.g., [56, 48]. Suppose that equality were definable in FOL, that is, that there existed some FOL specification in which a formula $Eq(x, y)$ could only be interpreted as equality in models. Then we could use such a formula to state that all models have singleton carriers: $\forall x. \forall y \,.\, Eq(x, y)$. However, FOL is not expressive enough to define models of fixed carrier size. In FOL, if a specification admits a model of non-empty carrier A then it also admits a model whose carrier is $A \cup \{b\}$, where b is some element that is not already in A. Indeed, pick some arbitrary element $a \in A$ and extend all the operations and predicates in the model to behave on b exactly the same as on a. Since the operation and predicate interpretations cannot distinguish between a and b, the model of carrier A and the

model of carrier $A \cup \{b\}$ satisfy exactly the same formulae. In particular, no FOL specification can admit only models of singleton carrier. One can define equivalence and congruence relations, but not actual equality. Since precise equality is sometimes desirable, extensions of FOL *with equality* have been proposed [56, 48], where a special binary predicate "=" is added to the logic together with axioms like equality introduction "$t = t$" and elimination "$(t_1 = t_2) \wedge \varphi[t_1/x] \rightarrow \varphi[t_2/x]$", and interpreted as the equality/identity relation in models.

Let us first discuss why we cannot use $\leftrightarrow$ as equality in matching logic. Indeed, since $M \models \varphi_1 \leftrightarrow \varphi_2$ iff $\overline{\rho}(\varphi_1) = \overline{\rho}(\varphi_2)$ for all $\rho : Var \to M$, one may be tempted to use $\leftrightarrow$ as equality. E.g., given a signature with one sort and one unary symbol f, one may think that the pattern $\exists y \,.\, f(x) \leftrightarrow y$ defines precisely the models where f is a function (because a function evaluates to only one value for any given argument, and the interpretation of variable pattern y has precisely one value). Unfortunately, that is not true. Consider model M with $M = \{1, 2\}$ and f_M the non-functional relation $f_M(1) = \{1, 2\}$, $f_M(2) = \emptyset$. Let $\rho : Var \to M$ be any M-valuation; recall (Definition 2.3) that ρ's extension $\overline{\rho}$ to patterns interprets "$\exists$" as union and "$\leftrightarrow$" as the complement of the symmetric difference. If $\rho(x) = 1$ then $\overline{\rho}(\exists y \,.\, f(x) \leftrightarrow y) = (M \backslash (\{1, 2\} \Delta \{1\})) \cup (M \backslash (\{1, 2\} \Delta \{2\})) = \{1, 2\} = M$. If $\rho(x) = 2$ then $\overline{\rho}(\exists y \,.\, f(x) \leftrightarrow y) = (M \backslash (\emptyset \Delta \{1\})) \cup (M \backslash (\emptyset \Delta \{2\})) = \{1, 2\} = M$. Hence, $M \models \exists y \,.\, f(x) \leftrightarrow y$, yet f_M is not a function, so $\leftrightarrow$ fails to capture the pattern equality.

The problem above is that the interpretation of $\varphi_1 \leftrightarrow \varphi_2$, depicted in Figure 4, is not two-valued ($\top$ or $\bot$), as we are used to think in classical logics. Specifically, $\overline{\rho}(\varphi_1) \neq \overline{\rho}(\varphi_2)$ does not suffice for $\overline{\rho}(\varphi_1 \leftrightarrow \varphi_2) = \emptyset$ to hold. Indeed, $\overline{\rho}(\varphi_1 \leftrightarrow \varphi_2) = M \setminus (\overline{\rho}(\varphi_1) \,\Delta\, \overline{\rho}(\varphi_2))$ and there is nothing to prevent, e.g., $\overline{\rho}(\varphi_1) \cap \overline{\rho}(\varphi_2) \neq \emptyset$, in which case $\overline{\rho}(\varphi_1) \,\Delta\, \overline{\rho}(\varphi_2) \neq M$. What we would like is a proper equality over patterns, $\varphi_1 = \varphi_2$, which behaves as a two-valued predicate: $\overline{\rho}(\varphi_1 = \varphi_2) = \emptyset$ when $\overline{\rho}(\varphi_1) \neq \overline{\rho}(\varphi_2)$, and $\overline{\rho}(\varphi_1 = \varphi_2) = M$ when $\overline{\rho}(\varphi_1) = \overline{\rho}(\varphi_2)$. Moreover, we want equalities to be used with patterns of any sort s_1 and in contexts of any sort s_2, similarly to the definedness and totality constructs in Section 5.1.

Equality can be defined quite compactly using the pattern totality and equivalence constructs, which were themselves defined using the assumed definedness symbols (Assumption 5.1, Section 5.1) and, respectively, the core $\wedge$ and $\neg$ constructs (Section 2). Specifically,

Notation 5.8. *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:*

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

Intuitively, $\varphi \leftrightarrow \varphi'$ matches the grey area in the diagram depicting pattern equivalence in Figure 4 (complement of the symmetric difference), so $\lfloor \varphi \leftrightarrow \varphi' \rfloor^{s_2}_{s_1}$ is interpreted as M_{s_2} iff the white area is empty, iff the two patterns match exactly the same elements. Formally,

Proposition 5.9. *Let $\varphi, \varphi' \in \text{PATTERN}_{s_1}$. Then:*

(1) *$\overline{\rho}(\varphi =^{s_2}_{s_1} \varphi') = \emptyset$ iff $\overline{\rho}(\varphi) \neq \overline{\rho}(\varphi')$, for any $\rho : Var \to M$*
(2) *$\overline{\rho}(\varphi =^{s_2}_{s_1} \varphi') = M_{s_2}$ iff $\overline{\rho}(\varphi) = \overline{\rho}(\varphi')$, for any $\rho : Var \to M$*
(3) *$M \models \varphi =^{s_2}_{s_1} \varphi'$ iff $M \models \varphi \leftrightarrow \varphi'$, for any model M*
(4) *$\models \varphi =^{s_2}_{s_1} \varphi'$ iff $\models \varphi \leftrightarrow \varphi'$*

Proof. Recall that $\varphi =^{s_2}_{s_1} \varphi'$ stands for $\lfloor \varphi \leftrightarrow \varphi' \rfloor^{s_2}_{s_1}$, which stands for $\neg \lceil \neg(\varphi \leftrightarrow \varphi') \rceil^{s_2}_{s_1}$.

(1) Therefore, $\overline{\rho}(\varphi =^{s_2}_{s_1} \varphi')$ is equal to $M_{s_2} \setminus (\lceil \_ \rceil^{s_2}_{s_1})_M(M_{s_1} \setminus (M_{s_1} \setminus (\overline{\rho}(\varphi_1) \,\Delta\, \overline{\rho}(\varphi_2))))$, which is further equal to $M_{s_2} \setminus (\lceil \_ \rceil^{s_2}_{s_1})_M(\overline{\rho}(\varphi_1) \,\Delta\, \overline{\rho}(\varphi_2))$. So $\overline{\rho}(\varphi =^{s_2}_{s_1} \varphi') = \emptyset$ iff $(\lceil \_ \rceil^{s_2}_{s_1})_M(\overline{\rho}(\varphi_1) \,\Delta\, \overline{\rho}(\varphi_2)) = M_{s_2}$, iff $\overline{\rho}(\varphi_1) \,\Delta\, \overline{\rho}(\varphi_2) \neq \emptyset$, iff $\overline{\rho}(\varphi) \neq \overline{\rho}(\varphi')$.

(2) Similarly to the above, we have $\overline{\rho}(\varphi =^{s_2}_{s_1} \varphi') = M_{s_2}$ iff $(\lceil \_ \rceil^{s_2}_{s_1})_M(\overline{\rho}(\varphi_1) \,\Delta\, \overline{\rho}(\varphi_2)) = \emptyset$, iff $\overline{\rho}(\varphi_1) \,\Delta\, \overline{\rho}(\varphi_2) = \emptyset$, iff $\overline{\rho}(\varphi) = \overline{\rho}(\varphi')$.
(3) $M \models \varphi =^{s_2}_{s_1} \varphi'$ iff $\overline{\rho}(\varphi =^{s_2}_{s_1} \varphi') = M_{s_2}$ for any $\rho : Var \to M$, iff $\overline{\rho}(\varphi) = \overline{\rho}(\varphi')$ for any $\rho : Var \to M$, iff (by Proposition 2.6) $M \models \varphi \leftrightarrow \varphi'$.
(4) $\models \varphi =^{s_2}_{s_1} \varphi'$ iff $M \models \varphi =^{s_2}_{s_1} \varphi'$ for any model M, iff (by the above) $M \models \varphi \leftrightarrow \varphi'$ for any model M, iff $\models \varphi \leftrightarrow \varphi'$.

Therefore, pattern equality satisfies all these properties. □

Note that (4) in the proposition above is not in conflict with the discussion at the beginning of this section concluding that we cannot use equivalence instead of equality. The example there illustrated an equivalence which was nested under a quantifier $(\exists y \,.\, f(x) \leftrightarrow y)$, while (4) above says that equivalence and equality are interchangeable at the pattern top.

Like for definedness and totality (Section 5.1), where we decided to drop the sorts s_1 and s_2 from $\lceil \_ \rceil^{s_2}_{s_1}$ and instead write $\lceil \_ \rceil$ because the sort of the enclosed pattern and that of the context dictate s_1 and s_2, we also take the freedom to drop the sort embellishments of $=^{s_2}_{s_1}$ and instead write just $=$. Like for definedness and totality, s_1 and s_2 can typically be inferred from context, and, if ambiguity arises, then we assume all instances. For example, "$x = x$" means "$x =^{s_2}_{s_1} x$" for any $s_1, s_2 \in S$ and $x \in Var_{s_1}$. Note that the equality symbol in algebraic specifications and in FOL (with equality) is also implicitly indexed by the sort of the two terms, although that sort is typically not mentioned as subscript; but one needs to exercise more care in matching logic, because equality patterns can be nested now. For example, the pattern in Section 9.2 defining linked list data-structures within maps,

$$list(x) = (x = 0 \wedge emp \vee \exists z \,.\, x \mapsto z * list(z))$$

is a sugared variant of the explicit patterns (one for each "equality context" sort s),

$$list(x) =^{s}_{Map} (x =^{Map}_{Nat} 0 \wedge emp \vee \exists z \,.\, x \mapsto z * list(z))$$

To minimize the number of disambiguation parentheses, we assumed that equality ($=$) binds tighter than conjunction ($\wedge$). We also assume that negation ($\neg$) binds tighter than equality ($=$). To avoid confusion, we may use disambiguation parentheses even if not strictly needed.

Despite the fact that patterns evaluate to any set of values and thus are more general than both terms (which evaluate to only one value) and predicates (which evaluate to one of two values), and despite the fact that Boolean combinations of patterns and quantification yield other patterns which can be used under any symbol in Σ, as we saw in Proposition 2.8, the proof rule/axiom schemas of (pure) predicate logic continue to be sound for matching logic. Now that we have equality, a natural question is whether the equality proof rule/axiom schemas of FOL with equality [56, 48] are also sound. For example, in FOL with equality, "equality elimination" states that terms can be substituted with equal terms in any context. A similar result holds for matching logic, where terms are replaced with arbitrary patterns:

Proposition 5.10. *The following hold:*

(1) *Equality introduction:* $\models \varphi = \varphi$
(2) *Equality elimination:* $\models (\varphi_1 = \varphi_2) \wedge \varphi[\varphi_1/x] \to \varphi[\varphi_2/x]$

Proof. (1) follows by (4) in Proposition 5.9 and by Proposition 3.1. For (2), let M be some model and $\rho : Var \to M$. By Proposition 2.6, it suffices to show $\overline{\rho}(\varphi_1 = \varphi_2) \cap \overline{\rho}(\varphi[\varphi_1/x]) \subseteq \overline{\rho}(\varphi[\varphi_2/x])$. If $\overline{\rho}(\varphi_1) \neq \overline{\rho}(\varphi_2)$ then $\overline{\rho}(\varphi_1 = \varphi_2) = \emptyset$ by Proposition 5.9, so the inclusion holds. Now suppose that $\overline{\rho}(\varphi_1) = \overline{\rho}(\varphi_2)$, which implies $\overline{\rho}(\varphi_1 = \varphi_2) = M$ by Proposition 5.9, so it suffices to show $\overline{\rho}(\varphi[\varphi_1/x]) \subseteq \overline{\rho}(\varphi[\varphi_2/x])$. The stronger result $\overline{\rho}(\varphi[\varphi_1/x]) = \overline{\rho}(\varphi[\varphi_2/x])$ in

fact holds, because the first element is a function of $\overline{\rho}(\varphi_1)$, the second element is the same function but of $\overline{\rho}(\varphi_2)$, and $\overline{\rho}(\varphi_1) = \overline{\rho}(\varphi_2)$. □

Notation 5.11. *From here on in the rest of the paper we write $\varphi \neq \varphi'$ instead of $\neg(\varphi = \varphi')$.*

One may wonder what really made it possible to define equality in matching logic, which is not possible in predicate or first-order logic. Let us consider the simplest instance of equality, $x = y$ between two variables, which is sugar for $\neg\lceil\neg(x \leftrightarrow y)\rceil$. After all, definedness-like predicates can also be defined in predicate logic; following the translation in Section 10, for example, the unary matching logic symbols $\lceil \_ \rceil$ are associated binary predicates $\pi_{\lceil \_ \rceil}$, and the definedness pattern axioms $\lceil x \rceil$ are translated into formula axioms $\pi_{\lceil \_ \rceil}(x, r)$. So the definedness symbol is not the key. The key is the capability to allow logical connectives between "terms", which is not allowed in first-order logic. For example, $x \leftrightarrow y$ already tells us whether x and y are interpreted as the same value or not: for any valuation ρ, it is indeed the case that $\overline{\rho}(x \leftrightarrow y)$ is the total set iff $\rho(x) = \rho(y)$ (see Proposition 2.6).

Equality elimination (Proposition 5.10) allows us to replace patterns by equal patterns in any context. Further, Proposition 5.9 allows us to replace any top-level $\leftrightarrow$ with $=$. In particular, the equivalences in Proposition 2.11 become $\models C_{\sigma,i}[\varphi_i \vee \varphi_i'] = C_{\sigma,i}[\varphi_i] \vee C_{\sigma,i}[\varphi_i']$ and $\models C_{\sigma,i}[\exists x \,.\, \varphi_i] = \exists x \,.\, C_{\sigma,i}[\varphi_i]$, respectively, meaning that we can propagate disjunction and existential quantification through symbols in any context, not only at the top level. Because of the stronger nature of equality, from here on we state properties in terms of equality instead of $\leftrightarrow$ whenever possible. Below is an important such property:

Proposition 5.12. ***(Constraint propagation)*** *Assume the same hypothesis as in Proposition 2.11: $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ and $\varphi_i \in \text{PATTERN}_{s_i}$ for all $1 \leq i \leq n$, a particular $1 \leq i \leq n$, and let $C_{\sigma,i}[\Box]$ be the context $\sigma(\varphi_1, \ldots, \varphi_{i-1}, \Box, \varphi_{i+1}, \ldots \varphi_n)$. Then for any pattern φ:*

(1) $\models C_{\sigma,i}[\varphi_i \wedge \lceil\varphi\rceil] = C_{\sigma,i}[\varphi_i] \wedge \lceil\varphi\rceil$
(2) $\models C_{\sigma,i}[\varphi_i \wedge \lfloor\varphi\rfloor] = C_{\sigma,i}[\varphi_i] \wedge \lfloor\varphi\rfloor$
(3) $\models C_{\sigma,i}[\varphi_i \wedge [\varphi]] = C_{\sigma,i}[\varphi_i] \wedge [\varphi]$ *if φ is a predicate (Definition 2.9 and Notation 5.7).*

Proof. We only show (1), because (2) and (3) are similar. Let $\rho : Var \to M$ and let $\overline{\rho}(C_{\sigma,i}) : M_{s_i} \to \mathcal{P}(M_s)$ be defined as $\overline{\rho}(C_{\sigma,i})(a) = \sigma_M(\overline{\rho}(\varphi_1), \ldots, \overline{\rho}(\varphi_{i-1}), a, \overline{\rho}(\varphi_{i+1}), \ldots, \overline{\rho}(\varphi_n))$. Then $\overline{\rho}(C_{\sigma,i}[\varphi_i \wedge \lceil\varphi\rceil]) = \overline{\rho}(C_{\sigma,i})(\overline{\rho}(\varphi_i) \cap \overline{\rho}(\lceil\varphi\rceil))$ and $\overline{\rho}(C_{\sigma,i}[\varphi_i] \wedge \lceil\varphi\rceil) = \overline{\rho}(C_{\sigma,i})(\overline{\rho}(\varphi_i)) \cap \overline{\rho}(\lceil\varphi\rceil])$. By Proposition 5.2, $\overline{\rho}(\lceil\varphi\rceil])$ is either the empty set or the total set, regardless of the result sort context (s_i vs. s). If the empty set, then $\overline{\rho}(C_{\sigma,i}[\varphi_i \wedge \lceil\varphi\rceil]) = \overline{\rho}(C_{\sigma,i})(\overline{\rho}(\varphi_i) \cap \emptyset) = \emptyset$ and $\overline{\rho}(C_{\sigma,i}[\varphi_i] \wedge \lceil\varphi\rceil) = \overline{\rho}(C_{\sigma,i})(\overline{\rho}(\varphi_i)) \cap \emptyset = \emptyset$. If the total set, then $\overline{\rho}(C_{\sigma,i}[\varphi_i \wedge \lceil\varphi\rceil]) = \overline{\rho}(C_{\sigma,i})(\overline{\rho}(\varphi_i) \cap M_{s_i}) = \overline{\rho}(C_{\sigma,i})(\overline{\rho}(\varphi_i))$ and $\overline{\rho}(C_{\sigma,i}[\varphi_i] \wedge \lceil\varphi\rceil) = \overline{\rho}(C_{\sigma,i})(\overline{\rho}(\varphi_i)) \cap M_s = \overline{\rho}(C_{\sigma,i})(\overline{\rho}(\varphi_i))$. Therefore, $\overline{\rho}(C_{\sigma,i}[\varphi_i \wedge \lceil\varphi\rceil]) = \overline{\rho}(C_{\sigma,i}[\varphi_i] \wedge \lceil\varphi\rceil)$. □

Constraint propagation allows us to propagate, through symbols, any logical constraints that appear in a conjunctive context. Indeed, as seen in the rest of this section (in particular in Section 5.8) and in Section 7, domain constraints can be expressed as equalities or as FOL predicates, and both of these are instances of matching logic predicates. Recall from Definition 2.9 that (matching logic) predicates are patterns which interpret to either the empty or the total set of their carrier. The definedness symbol applied to a predicate, the square brackets in $[\varphi]$ (Notation 5.7), does not change the semantics of the predicate, but thanks to its polymorphic nature (Notation 5.6) we can syntactically move φ from the sort context of the argument pattern (s_i) of σ to the sort context of σ's result (s).

Proposition 5.12 (constraint propagation) and Proposition 2.10 (structural framing) are particularly useful to localize proof efforts, as illustrated in the example in Section 2.2.

5.3. **Membership.** Since in matching logic a pattern φ evaluates to a set of values while a variable (pattern) x evaluates to just a (set containing only one) value, the membership question, "does $x \in \varphi$ hold?", is natural. As seen later in Section 11, membership in fact plays a key role in proving the completeness of matching logic reasoning. Fortunately, membership can be quite easily defined as a derived construct in matching logic, making use of the definedness symbol (Section 5.1), in a similar way to equality (Section 5.2):

Notation 5.13. *If $x \in Var_{s_1}$, $\varphi \in \text{PATTERN}_{s_1}$ and $s_2 \in S$, then we introduce the notation*

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

Like for definedness, totality and equality, there is a membership construct for each pair of sorts s_1 (for variable and pattern) and s_2 (for context); we take the freedom to omit them.

Proposition 5.14. *With the above, the following hold:*

(1) $\overline{\rho}(x \in^{s_2}_{s_1} \varphi) = \emptyset$ *iff* $\rho(x) \notin \overline{\rho}(\varphi)$, *for any* $\rho : Var \to M$
(2) $\overline{\rho}(x \in^{s_2}_{s_1} \varphi) = M_{s_2}$ *iff* $\rho(x) \in \overline{\rho}(\varphi)$, *for any* $\rho : Var \to M$
(3) $\models (x \in^{s_2}_{s_1} \varphi) =^{s_3}_{s_2} (x \wedge \varphi =^{s_2}_{s_1} x)$, *for any sort* s_3

Proof. Recall that $x \in^{s_2}_{s_1} \varphi$ is $[x \wedge \varphi]^{s_2}_{s_1}$.

(1) Therefore, we have $\overline{\rho}(x \in^{s_2}_{s_1} \varphi) = (\lceil \_ \rceil^{s_2}_{s_1})_M(\{\rho(x)\} \cap \overline{\rho}(\varphi))$, so $\overline{\rho}(x \in^{s_2}_{s_1} \varphi) = \emptyset$ iff $\{\rho(x)\} \cap \overline{\rho}(\varphi) = \emptyset$, that is, iff $\rho(x) \notin \overline{\rho}(\varphi)$.
(2) Similarly to above, $\overline{\rho}(x \in^{s_2}_{s_1} \varphi) = M_{s_2}$ iff $\{\rho(x)\} \cap \overline{\rho}(\varphi) \neq \emptyset$, that is, iff $\rho(x) \in \overline{\rho}(\varphi)$.
(3) Let M be some model and $\rho : Var \to M$. By Proposition 5.9, the property holds iff we can show $\overline{\rho}(x \in^{s_2}_{s_1} \varphi) = \overline{\rho}(x \wedge \varphi =^{s_2}_{s_1} x)$. Since the membership and equality patterns are predicates (Definition 2.4), and thus they evaluate either to the entire set or to the empty set, the following completes the proof: by (2) we have $\overline{\rho}(x \in^{s_2}_{s_1} \varphi) = M_{s_2}$ iff $\rho(x) \in \overline{\rho}(\varphi)$, iff $\{\rho(x)\} \cap \overline{\rho}(\varphi) = \{\rho(x)\}$, iff, by (2) in Proposition 5.9, $\overline{\rho}(x \wedge \varphi =^{s_2}_{s_1} x) = M_{s_2}$; and by (1) we have $\overline{\rho}(x \in^{s_2}_{s_1} \varphi) = \emptyset$ iff $\rho(x) \notin \overline{\rho}(\varphi)$, iff $\{\rho(x)\} \cap \overline{\rho}(\varphi) \neq \{\rho(x)\}$, iff, by (1) in Proposition 5.9, $\overline{\rho}(x \wedge \varphi =^{s_2}_{s_1} x) = \emptyset$;

Therefore, these basic properties hold. □

Property (3) in Proposition 5.14 suggests that the equality $x \wedge \varphi = x$ can be regarded as an alternative definition of membership $x \in \varphi$, but we prefer $\lceil x \wedge \varphi \rceil$ because is simpler (the other one requires an additional sort, s_3, for the context of the equality).

Proposition 2.8 showed that some of the proof rule/axiom schemas of FOL with equality are already sound for matching logic, namely the rules corresponding to (pure) predicate logic. Proposition 5.10 further showed that the equality-related rules/axioms are also sound. The soundness of several other rule/axiom schemas are shown below, essentially completing the soundness of the matching logic proof system (discussed later in Section 11), except for one rule, Substitution, which needs more discussion and we postpone it to Section 11:

Proposition 5.15. *The following hold:*

(1) $\models \forall x \,.\, x \in \varphi$ *iff* $\models \varphi$
(2) $\models (x \in y) = (x = y)$ *when* $x, y \in Var$
(3) $\models (x \in \neg\varphi) = \neg(x \in \varphi)$
(4) $\models (x \in \varphi_1 \wedge \varphi_2) = (x \in \varphi_1) \wedge (x \in \varphi_2)$
(5) $\models (x \in \exists y.\varphi) = \exists y.(x \in \varphi)$, *with* x *and* y *distinct*
(6) $\models x \in \sigma(\varphi_1, ..., \varphi_{i-1}, \varphi_i, \varphi_{i+1}, ..., \varphi_n) = \exists y.(y \in \varphi_i \wedge x \in \sigma(\varphi_1, ..., \varphi_{i-1}, y, \varphi_{i+1}, ..., \varphi_n))$

Proof. The proofs below make repetitive use of Propositions 5.9 and 5.14:

(1) Let M be a model. Then $M \models \forall x\,.\,x \in \varphi$ iff $M \models x \in \varphi$ (Proposition 2.6), iff $\overline{\rho}(x \in \varphi) = M$ for any $\rho : Var \to M$, iff $\rho(x) \in \overline{\rho}(\varphi)$ for any $\rho : Var \to M$, iff $\overline{\rho}(\varphi) = M$ for any $\rho : Var \to M$, iff $M \models \varphi$.
(2) It suffices to show $\overline{\rho}(x \in y) = M$ iff $\overline{\rho}(x = y) = M$ for any model M and any $\rho : Var \to M$, that is, that $\rho(x) \in \{\rho(y)\}$ iff $\rho(x) = \rho(y)$, which obviously holds.
(3) It suffices to show $\overline{\rho}(x \in \neg\varphi) = M$ iff $\overline{\rho}(x \in \varphi) = \emptyset$ for any model M and any $\rho : Var \to M$, that is, that $\rho(x) \in M \backslash \overline{\rho}(\varphi)$ iff $\rho(x) \notin \overline{\rho}(\varphi)$, which obviously holds.
(4) It suffices to show $\rho(x) \in \overline{\rho}(\varphi_1) \cap \overline{\rho}(\varphi_2)$ iff $\rho(x) \in \overline{\rho}(\varphi_1)$ and $\rho(x) \in \overline{\rho}(\varphi_2)$ for any model M and any $\rho : Var \to M$, which obviously holds.
(5) It suffices to show for any model M and any $\rho : Var \to M$, that $\rho(x) \in \bigcup\{\overline{\rho'}(\varphi) \mid \rho' : Var \to M,\ \rho'{\restriction}_{Var\backslash\{y\}} = \rho{\restriction}_{Var\backslash\{y\}}\}$ iff $\bigcup\{\overline{\rho'}(x \in \varphi) \mid \rho' : Var \to M,\ \rho'{\restriction}_{Var\backslash\{y\}} = \rho{\restriction}_{Var\backslash\{y\}}\} = M$. It is easy to see that each of the two statements holds iff there exists some $\rho' : Var \to M$ with $\rho'{\restriction}_{Var\backslash\{y\}} = \rho{\restriction}_{Var\backslash\{y\}}$ such that $\rho(x) \in \overline{\rho'}(\varphi)$.
(6) It suffices to prove for any model M and any valuation $\rho : Var \to M$, that

$$\rho(x) \in \sigma_M(\overline{\rho}(\varphi_1), \ldots, \overline{\rho}(\varphi_{i-1}), \overline{\rho}(\varphi_i), \overline{\rho}(\varphi_{i+1}), \ldots, \overline{\rho}(\varphi_n))$$

iff there exists a $\rho' : Var \to M$ with $\rho'{\restriction}_{Var\backslash\{y\}} = \rho{\restriction}_{Var\backslash\{y\}}$ such that $\rho'(y) \in \overline{\rho}(\varphi_i)$ and

$$\rho(x) \in \sigma_M(\overline{\rho}(\varphi_1), \ldots, \overline{\rho}(\varphi_{i-1}), \{\rho'(y)\}, \overline{\rho}(\varphi_{i+1}), \ldots, \overline{\rho}(\varphi_n)),$$

which obviously holds.

The proof is complete. □

We next define several common relations using patterns, such as functions.

5.4. **Functions.** Matching logic makes no distinction between function and predicate symbols, treating all symbols uniformly as pattern symbols which are interpreted relationally. A natural question is whether there is any way, in matching logic, to state that a symbol is to be interpreted as a function in all models. We show a more general result, namely that there is a way to state that any pattern, not only a symbol, has a functional interpretation.

Definition 5.16. *Pattern φ is **functional in a model** M iff $|\overline{\rho}(\varphi)| = 1$ for any valuation $\rho : Var \to M$. Furthermore, φ is **functional in** $F \subseteq$ Pattern, or simply **functional** when F is understood, iff it is functional in all models M with $M \models F$.*

Recall from the preamble of Section 5 that (S, Σ, F) was assumed to be an arbitrary but fixed matching logic specification. Therefore F is understood, so we take the freedom to just say "φ is functional" instead of "φ is functional in F".

The following trivial result relates functional patterns to (total) functions:

Proposition 5.17. *If $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ and M is a Σ-model, then pattern $\sigma(x_1, \ldots, x_n)$ is functional in M iff $\sigma_M : M_{s_1} \times \cdots \times M_{s_n} \to M_s$ is a total function in M, that is, iff $\sigma_M(a_1, ..., a_n)$ contains precisely one element for any elements $a_1 \in M_{s_1}$, ..., $a_n \in M_{s_n}$.*

Proof. Pattern $\sigma(x_1, \ldots, x_n)$ is functional in M iff $|\overline{\rho}(\sigma(x_1, \ldots, x_n))| = 1$ for any valuation $\rho : Var \to M$ (by Definition 5.16), iff $|\sigma_M(a_1, ..., a_n)| = 1$ for any $a_1 \in M_{s_1}$, ..., $a_n \in M_{s_n}$. □

The following proposition gives an axiomatic characterization of functional patterns:

Proposition 5.18. *Pattern φ is functional in model M iff $M \models \exists y \,.\, (\varphi = y)$, where variable y is chosen so that $y \notin FV(\varphi)$. Therefore, φ is functional iff $F \models \exists y \,.\, (\varphi = y)$.*

Proof. φ is functional in M iff $|\overline{\rho}(\varphi)| = 1$ for any $\rho : Var \to M$ (by Definition 5.16), iff for any $\rho : Var \to M$ there is some $a \in M$ such that $\overline{\rho}(\varphi) = \{a\}$, iff for any $\rho : Var \to M$ there is some $\rho' : Var \to M$ with $\rho' {\restriction}_{Var \setminus \{y\}} = \rho {\restriction}_{Var \setminus \{y\}}$ such that $\overline{\rho'}(\varphi) = \overline{\rho'}(y)$ (by (1) in Proposition 2.6), iff $M \models \exists y \,.\, (\varphi = y)$ (by Definition 2.3 and Proposition 5.9). □

Corollary 5.19. *Variables are functional: $\models \exists y \,.\, x = y$ for any variable x.*

Proof. Immediate consequence of Definition 5.16 and Proposition 5.18, because variables are interpreted as singletons: $\overline{\rho}(x) = \{\rho(x)\}$ for any valuation $\rho : Var \to M$. □

We have seen in the discussion at the beginning of Section 5.2 that if f is a one-argument symbol, the pattern $\exists y \,.\, f(x) \leftrightarrow y$ is not strong enough to enforce $f(x)$ to be functional. However, thanks to Proposition 5.18, using equality instead of equivalence works:

Corollary 5.20. *If $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ and M is a Σ-model, then σ_M is a total function iff $M \models \exists y \,.\, \sigma(x_1, \ldots, x_n) = y$.*

Proof. By Propositions 5.17 and 5.18. □

Hence, we can state that a symbol $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ is a function in all models by requiring $\sigma(x_1, \ldots, x_n)$ to be a functional pattern, which by Proposition 5.18 is equivalent to stating that the pattern $\exists y \,.\, \sigma(x_1, \ldots, x_n) = y$ holds (i.e., it is entailed by F), where x_1, ..., x_n are free variables. The simplest way to ensure this is to add this pattern directly to F, as an axiom. To avoid manually writing such trivial pattern axioms for lots of symbols which are meant to be interpreted as functions, we adopt the following notation and terminology:

Definition 5.21. *For a symbol $\sigma \in \Sigma_{s_1 \ldots s_n, s}$, the notation*

$$\sigma : s_1 \times \cdots \times s_n \to s$$

*is syntactic sugar for stating that F contains the pattern $\exists y \,.\, \sigma(x_1, \ldots, x_n) = y$. If $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ is a symbol such that $\sigma : s_1 \times \cdots \times s_n \to s$, then we call σ a **function symbol**. Patterns built with only function symbols are called **term patterns**, or simply just **terms**.*

Definition 5.21 is instrumental to capturing algebraic specifications and first-order logic as instances of matching logic; full details are given in Sections 6 and 7.

Corollary 5.22. *Term patterns are functional: $\models \exists y \,.\, t = y$ for any term pattern t.*

Proof. Structural induction on term pattern t. Obvious when t is a variable. Let t be $\sigma(t_1, \ldots, t_n)$ with $\sigma : s_1 \times \ldots \times s_n \to s$ and $t_1, \ldots, t_n$ term patterns of sorts s_1, ..., s_n, respectively, and let $\rho : Var \to M$. Then $\overline{\rho}(t) = \sigma_M(\overline{\rho}(t_1), \ldots, \overline{\rho}(t_n))$. By the induction hypothesis, $t_1, \ldots, t_n$ are functional, that is, $\overline{\rho}(t_1)$, ..., $\overline{\rho}(t_n)$ are singleton sets (Definition 5.16), say $\{m_1\}$, ..., $\{m_n\}$, respectively. Then $\overline{\rho}(t) = \sigma_M(m_1, \ldots, m_n)$, which is also a singleton set, say $\{m\}$, as σ_M is a function (Corollary 5.20). The rest follows by Proposition 5.18. □

In FOL, the Substitution axiom $((\forall x : s \,.\, \varphi) \to \varphi[t/x])$ allows for universally quantified variables to be substituted with any terms. Together with the proof rules and axioms of predicate logic, Substitution makes FOL deduction complete. An important property of term patterns in matching logic is that the Substitution axiom of FOL holds for them:

Corollary 5.23. *If φ is any pattern and t is a term pattern of sort s, then*

Term Substitution: $\models (\forall x : s \,.\, \varphi) \to \varphi[t/x]$

Proof. Let $\rho : Var \to M$ be any valuation. Then

$$\overline{\rho}(\forall x \,.\, \varphi) = \bigcap \{\overline{\rho'}(\varphi) \mid \rho' \restriction_{Var \setminus \{x\}} = \rho \restriction_{Var \setminus \{x\}}\} \subseteq \overline{\rho''}(\varphi)$$

where $\rho'' : Var \to M$ is such that $\rho'' \restriction_{Var \setminus \{x\}} = \rho \restriction_{Var \setminus \{x\}}$ and $\{\rho''(x)\} = \overline{\rho}(t)$. Such a ρ'' exists thanks to Corollary 5.22 and can only be $\rho'' = \rho[m/x]$ where $\overline{\rho}(t) = \{m\}$, so $\overline{\rho''}(\varphi) = \overline{\rho}(\varphi[t/x])$. □

Note Corollary 5.23 generalizes Corollary 5.22: pick φ to be $\exists y \,.\, x = y$; then $\forall x : s \,.\, \varphi$ is a tautology and $\varphi[t/x]$ is $\exists y \,.\, t = y$, which by Proposition 5.18 implies that t is functional. Corollary 5.23 also generalizes (5) in Proposition 2.8, because variables are particular terms.

Corollary 5.23, Proposition 5.10 and Proposition 2.8 imply that FOL reasoning with or without equality is sound for matching logic, provided that the Substitution axiom of FOL is only applied when t is a term pattern. To avoid confusing the FOL Substitution axiom schema with the matching logic variant in Corollary 5.23, we called the later Term Substitution. As shown in Section 11, Term Substitution can be generalized a bit into Functional Substitution, which takes functional patterns instead of term patterns t, but in general it is not sound for arbitrary patterns instead of t.

Since functional patterns evaluate to singleton sets for any valuation, the conjunction of two functional patterns evaluate either to the empty set when the two patterns evaluate to different singleton sets, or to the same singleton set when the two patterns evaluate to the same singleton set. Formally,

Proposition 5.24. *If φ and φ' are functional patterns of the same sort, then:*

(1) $\models ((\varphi \wedge \varphi') = \bot) = (\varphi \neq \varphi')$
(2) $\models ((\varphi \wedge \varphi') \neq \bot) = (\varphi = \varphi')$
(3) $\models (\varphi \wedge \varphi') = (\varphi \wedge (\varphi = \varphi'))$

Proof. Trivial: pick a model M and a valuation $\rho : Var \to M$, and apply the definitions. □

Particular functions with particular properties, such as injective of surjective functions, can be defined in a conventional way using conventional FOL. For example, pattern (one-argument functions only, for simplicity)

$$f(x) = f(y) \to x = y$$

states that f is injective and pattern

$$\exists x \,.\, f(x) = y$$

states that f is surjective. We only show the former: if $(M, f_M : M \to M)$ is any model satisfying $f(x) = f(y) \to x = y$, then f_M must be injective. Indeed, let $a, b \in M$ such that $a \neq b$ and $f_M(a) = f_M(b)$. Pick $\rho : Var \to M$ such that $\rho(x) = a$ and $\rho(y) = b$. Since M satisfies the axiom above, we get $\overline{\rho}(f(x) = f(y)) \subseteq \overline{\rho}(x = y)$. But Proposition 5.9 implies that $\overline{\rho}(x = y) = \emptyset$ and $\overline{\rho}(f(x) = f(y)) = M$, which is a contradiction. We can also show that

any model whose f is injective satisfies the axiom. Let $(M, f_M : M \to M)$ be any model such that f_M is injective. It suffices to show $\overline{\rho}(f(x) = f(y)) \subseteq \overline{\rho}(x = y)$ for any $\rho : Var \to M$, which follows by Proposition 5.9: if $\rho(x) = \rho(y)$ then $\overline{\rho}(f(x) = f(y)) = \overline{\rho}(x = y) = M$, and if $\rho(x) \neq \rho(y)$ then $\overline{\rho}(f(x) = f(y)) = \overline{\rho}(x = y) = \emptyset$ because f_M is injective.

With the notation $\varphi \neq \varphi'$ for $\neg(\varphi = \varphi')$ introduced in Section 5.2, $x \neq y \to f(x) \neq f(y)$ is an alternative way to capture the injectivity of f.

5.5. **Partial Functions.** In FOL, operation symbols are interpreted as *total* functions by default, meaning that they are defined on all the elements in their domain. Interpreting function symbols as partial functions leads to a completely different logic, called *partial FOL* in the literature (see, e.g., [35]), which has many different axioms to properly capture the desired properties of definedness and undefinedness. Our interpretations of symbols into powersets allows us not only to elegantly define definedness (Section 5.1), but also to define partial functions without a need to develop a different logic. Specifically, the pattern

$$\neg\sigma(x_1, \ldots, x_n) \quad \vee \quad \exists y \,.\, \sigma(x_1, \ldots, x_n) = y,$$

where $\neg\sigma(x_1, \ldots, x_n)$ can be equivalently replaced with $\sigma(x_1, \ldots, x_n) = \bot_s$, states that $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ is a partial function. From now on we use the notation (note the "$\rightharpoonup$" symbol)

$$\sigma : s_1 \times \cdots \times s_n \rightharpoonup s$$

to automatically assume a pattern like the above for σ. For example, a division partial function which is undefined iff the denominator is 0 can be specified as:

$$\begin{array}{l} \_ / \_ : Nat \times Nat \rightharpoonup Nat \\ x/y = \bot \leftrightarrow y = 0 \end{array}$$

which means a symbol $\_ / \_ \in \Sigma_{Nat \times Nat, Nat}$ with pattern axioms $\neg(x/y) \vee \exists z \,.\, x/y = z$ and $x/y = \bot \leftrightarrow y = 0$; the latter is equivalent to $\lceil x/y \rceil = (y \neq 0)$ and to $\lfloor \neg(x/y) \rfloor = (y = 0)$.

5.6. **Total Relations.** Recall from Section 5.4 that we can define total functions using patterns of the form $\exists y \,.\, \sigma(x_1, \ldots, x_n) = y$, stating not only that the interpretation of σ in model M, σ_M, is defined in any of its arguments, but also that it has only one value. We sometimes want to state that relations, not only functions, are total. All we have to do is to say that the relation is non-empty for any arguments, which can be easily stated with a pattern of the form $\lceil \sigma(x_1, \ldots, x_n) \rceil^s_s$, equivalent to $\sigma(x_1, \ldots, x_n) \neq \bot_s$. We write

$$\sigma : s_1 \times \cdots \times s_n \Rightarrow s$$

to automatically state that σ is a total relation.

5.7. **Constructors, Unification, Anti-Unification.** Constructors can be used to build programs, data, as well as semantic structures to define and reason about languages and programs. Hence, constructors have been extensively studied in the literature, using various approaches and logical formalisms. We believe that classic approaches to constructors can also be adapted to our matching logic setting, either directly by redefining the corresponding concepts (e.g., the matching logic analogous to initial algebras [44], etc.) or indirectly by translating them together with their underlying logic to matching logic (e.g., following the translations of FOL and algebraic specifications to matching logic in Sections 7 and 6, respectively). Here we discuss a different approach. Specifically, we show how the dual nature of patterns to specify both structure and constraints can make some of the definitions

and notions related to constructors more elegant and appealing. For example, unification and anti-unification can be seen as conjunction and, respectively, disjunction of patterns.

One main property of constructors is that they collectively can construct all the elements of their target domain; i.e., the target domain has "no junk" [44]. One simple pattern stating that a unary symbol f is to be interpreted as a surjective relation in every model is $\exists x \,.\, f(x)$. Generalizing this idea to an arbitrary set of n symbols

$$\{c_i \in \Sigma_{s_i^1 \ldots s_i^{m_i}, s_i} \mid 1 \leq i \leq n\}$$

that we want to be constructors for target sort s, we get the following "no junk" pattern:

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

For example, applied to the usual 0 and *succ* constructors of natural numbers, the above becomes $0 \;\vee\; \exists x : Nat \,.\, succ(x)$. Indeed, the interpretation of this pattern in the model of natural numbers is the entire domain; or said differently, any number matches this pattern.

The other main property of constructors is that they yield a unique way to construct each element in the target domain; i.e., the target domain has "no confusion" [44]. That means two separate types of conditions, each specifiable with patterns. First, each constructor c_i builds a set of elements distinct from that of any other constructor c_j with $s_j = s_i$:

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

Recall that, by convention, free variables in pattern axioms are assumed universally quantified. For our 0 and *succ* example, the above becomes $\neg(0 \wedge succ(x))$, stating that $succ(x)$ is different from 0 for any x. Second, each constructor c_i is injective in all its arguments at once (regarded as a tuple), which can be specified with a pattern as follows:

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

Indeed, the above pattern ensures that in any model M, if

$$(c_i)_M(a_i^1, \ldots, a_i^{m_i}) \cap (c_i)_M(b_i^1, \ldots, b_i^{m_i}) \neq 0$$

then it must be that $a_i^1 = b_i^1$, ..., $a_i^{m_i} = b_i^{m_i}$.

Putting all the above together, below we formally introduce constructors:

Definition 5.25. *Given a specification (S, Σ, F), the symbols in set*

$$\{c_i \in \Sigma_{s_i^1 \ldots s_i^{m_i}, s_i} \mid 1 \leq i \leq n\}$$

*are called **constructors** iff they have the following properties:*

No junk: *For any sort $s \in \{s_i \mid 1 \leq i \leq n\}$,*

$$F \models \bigvee_{s_i = s} \exists x_i^1 : s_i^1 \ldots \exists x_i^{m_i} : s_i^{m_i} \,.\, c_i(x_i^1, \ldots, x_i^{m_i})$$

No confusion, different constructors: *For any $1 \leq i \neq j \leq n$ with $s_j = s_i$,*

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

No confusion, same constructor: *For any $1 \leq i \leq n$,*

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

*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 F include all the patterns above.*

It is easy to see that if the symbol σ that occurs in the context $C_{\sigma,i}[\Box]$ in Definition 2.12 is a constructor, then $C_{\sigma,i}[\Box]$ is injective, and thus, by Propositions 2.13 and 2.11, it enjoys full distributivity w.r.t. the matching logic constructs $\wedge$, $\vee$, $\forall$ and $\exists$. Thanks to Propositions 5.9 and 5.10, we can therefore apply these distributivity properties of constructors in any context. In addition to the distributivity properties, the following equality properties of constructors turned out to also be very useful in program verification efforts:

Proposition 5.26. *Given a set of constructors $\{c_i \in \Sigma_{s_i^1 \ldots s_i^{m_i}, s_i} \mid 1 \leq i \leq n\}$ for a specification (S, Σ, F), the following hold:*

Case analysis: *If φ is a pattern of sort $s \in \{s_i \mid 1 \leq i \leq n\}$, then*

$$F \models \varphi = \bigvee_{s_i = s} \exists x_i^1 : s_i^1 \ldots \exists x_i^{m_i} : s_i^{m_i} \ . \ \varphi \wedge c_i(x_i^1, \ldots, x_i^{m_i})$$

Additionally, if the constructors and φ are all functional, then

$$F \models \varphi = \bigvee_{s_i = s} \exists x_i^1 : s_i^1 \ldots \exists x_i^{m_i} : s_i^{m_i} \ . \ c_i(x_i^1, \ldots, x_i^{m_i}) \wedge (\varphi = c_i(x_i^1, \ldots, x_i^{m_i}))$$

Different constructors: *If $1 \leq i \neq j \leq n$ with $s_i = s_j$, and $\varphi_i^1 \in \text{PATTERN}_{s_i^1}$, ..., $\varphi_i^{m_i} \in \text{PATTERN}_{s_i^{m_i}}$, and $\varphi_j^1 \in \text{PATTERN}_{s_j^1}$, ..., $\varphi_j^{m_j} \in \text{PATTERN}_{s_j^{m_j}}$, then*

$$F \models c_i(\varphi_i^1, \ldots, \varphi_i^{m_i}) \wedge c_j(\varphi_j^1, \ldots, \varphi_j^{m_j}) = \bot$$

Same constructor: *If $\varphi_i^1, \psi_i^1 \in \text{PATTERN}_{s_i^1}$, ..., $\varphi_i^{m_i}, \psi_i^{m_i} \in \text{PATTERN}_{s_i^{m_i}}$, then*

$$F \models c_i(\varphi_i^1, \ldots, \varphi_i^{m_i}) \wedge c_i(\psi_i^1, \ldots, \psi_i^{m_i}) = c_i(\varphi_i^1 \wedge \psi_i^1, \ldots, \varphi_i^{m_i} \wedge \psi_i^{m_i})$$

Proof. Case analysis follows by Proposition 5.9, which reduces equality ($=$) to double implication ($\leftrightarrow$). The latter follows by the propositional distributivity of $\wedge$ over $\vee$ and, of course, the "no junk" requirement of constructors in Definition 5.25. The part where the constructors and φ are functional is an immediate corollary, making use of Proposition 5.24.

Different constructors: Suppose that there is some model M and valuation $\rho : \mathit{Var} \to M$ such that $\overline{\rho}(c_i(\varphi_i^1, \ldots, \varphi_i^{m_i}) \wedge c_j(\varphi_j^1, \ldots, \varphi_j^{m_j})) \neq \emptyset$. Then there are some elements $a_i^1 \in \overline{\rho}(\varphi_i^1)$, ..., $a_i^{m_i} \in \overline{\rho}(\varphi_i^{m_i})$, and $a_j^1 \in \overline{\rho}(\varphi_j^1)$, ..., $a_j^{m_j} \in \overline{\rho}(\varphi_j^{m_j})$ such that $(c_i)_M(a_i^1, \ldots, a_i^{m_i}) \cap (c_j)_M(a_j^1, \ldots, a_j^{m_j}) \neq \emptyset$, which contradicts the "no confusion" requirement for different constructors in Definition 5.25.

Same constructor: By Proposition 5.9, we can replace $=$ with $\leftrightarrow$. The $\leftarrow$ implication is immediate by structural framing, Proposition 2.10. For the $\rightarrow$ implication, let M be a model, $\rho : \mathit{Var} \to M$ a valuation, and $b \in \overline{\rho}(c_i(\varphi_i^1, \ldots, \varphi_i^{m_i}) \wedge c_i(\psi_i^1, \ldots, \psi_i^{m_i}))$. Then there are some elements $u_i^1 \in \overline{\rho}(\varphi_i^1)$, ..., $u_i^{m_i} \in \overline{\rho}(\varphi_i^{m_i})$, and $v_i^1 \in \overline{\rho}(\psi_i^1)$, ..., $v_i^{m_i} \in \overline{\rho}(\psi_i^{m_i})$ such that $b \in (c_i)_M(u_i^1, \ldots, u_i^{m_i}) \wedge (c_i)_M(v_i^1, \ldots, v_i^{m_i}))$, that is, $b \in \overline{\rho'}(c_i(x_i^1, \ldots, x_i^{m_i}) \wedge c_i(y_i^1, \ldots, y_i^{m_i}))$, where $\rho' : \mathit{Var} \to M$ is some valuation that takes x_i^1 to u_i^1, ..., $x_i^{m_i}$ to $u_i^{m_i}$, and y_i^1 to v_i^1, ..., $y_i^{m_i}$ to $v_i^{m_i}$. By the "no confusion" requirement for the same constructor in Definition 5.25 we conclude that $b \in \overline{\rho'}(c_i(x_i^1 \wedge y_i^1, \ldots, x_i^{m_i} \wedge y_i^{m_i}))$, that is, $b \in (c_i)_M(\{u_i^1\} \wedge \{v_i^1\}, \ldots, \{u_i^{m_i}\} \wedge \{v_i^{m_i}\})$. This can only happen when $u_i^1 = v_i^1$, ..., $u_i^{m_i} = v_i^{m_i}$. And in that case it can only be that $b \in \overline{\rho}(c_i(\varphi_i^1 \wedge \psi_i^1, \ldots, \varphi_i^{m_i} \wedge \psi_i^{m_i}))$. $\Box$

The case analysis property is useful when additional constraints are needed on a pattern in order to reason with it. For example, if b is a Boolean (symbolic) expression in a given positive context, $C[b]$, then we can replace b with $\mathit{true} \wedge (b = \mathit{true}) \vee \mathit{false} \wedge (b = \mathit{false})$, and then by Propsitions 2.11 (distributivity) and 5.12 (constraint propagation) we can reduce $C[b]$ to $C[\mathit{true}] \wedge (b = \mathit{true}) \vee C[\mathit{false}] \wedge (b = \mathit{false})$. Similarly, if e is a Nat (symbolic) expression in a positive context $C[e]$, then we can reduce the pattern $C[e]$ to the pattern $C[0] \wedge (e = 0) \vee \exists x \,.\, C[\mathit{succ}(x)] \wedge (e = \mathit{succ}(x))$, which may be further reducible. The other two properties in Proposition 5.26 are self-explanatory and clearly useful for the same reasons why constructors are useful, but we found them particularly useful when attempting to do symbolic execution using the operational semantics rules of a language. As explained in [28], the main technical instrument there is unification: indeed, in order to check if a symbolic program configuration φ can be executed with an operational rule $\mathit{left} \Rightarrow \mathit{right}$, a unification of φ and left is attempted. If it fails then the rule cannot be applied; if it succeeds then the rule can be applied and φ is advanced to $\mathit{right} \wedge \psi$, where ψ is the constraints resulting from unifying φ and left. As discussed below, unification can be achieved in matching logic by pattern conjunction, which makes the last properties in Proposition 5.12 indispensable.

We next show how unification and anti-unification can be explained as conjunction and, respectively, disjunction of matching logic patterns. To fall into the conventional setting, for the reminder of this section assume that all the symbols are functional constructors and all the starting patterns are term patterns (Definition 5.21) built with such constructors.

Let us re-think unification in terms of matching logic and pattern matching. Consider two patterns φ_1 and φ_2 having the same sort. Each of them is matched by a potentially infinite set of elements. We are interested in the elements that match both φ_1 and φ_2. Moreover, we are interested in some pattern φ that captures all these elements. Clearly $\varphi \to \varphi_1$ and $\varphi \to \varphi_2$, and we would like the most general such φ, that is, if $\varphi' \to \varphi_1$ and $\varphi' \to \varphi_2$ then $\varphi' \to \varphi$. We do not have to search for such a φ any further, because it is, by definition, $\varphi_1 \wedge \varphi_2$. All we have to do then is to simplify $\varphi_1 \wedge \varphi_2$ to a convenient form, say a term pattern constrained with equalities telling how φ_1 and φ_2 fall as instances, using matching logic reasoning. Let us exemplify this when $\varphi_1 \equiv f(g(x), x)$ and $\varphi_2 \equiv f(y, 0)$ where f is a binary symbol (functional construct), g is unary, and 0 is a constant:

$$
\begin{array}{rcl}
f(g(x), x) \wedge f(y, 0) & = & \text{(by Proposition 5.26)} \\
f(g(x) \wedge y, x \wedge 0) & = & \text{(by Proposition 5.24)} \\
f(g(x) \wedge (y = g(x)), x \wedge (x = 0)) & = & \text{(by Proposition 5.12)} \\
f(g(x), x) \wedge (y = g(x)) \wedge (x = 0) & = & \text{(by Proposition 5.10)} \\
f(g(0), 0) \wedge (y = g(0)) \wedge (x = 0) & &
\end{array}
$$

Therefore, using matching logic reasoning we obtained both the most general unifier of the two term patterns, encoded as a conjunction of equalities $(y = g(0)) \wedge (x = 0)$, and the unifying term pattern $f(g(0), 0)$. We believe that unification algorithms should not be difficult to adapt into matching logic proof search heuristics capable of producing proofs like above, thus narrowing the gap between tools and certifiable program verification.

Anti-unification, or generalization, can be dually regarded as disjunction of patterns[2]. Let us briefly recall Plotkin's original two-rule algorithm [76]:

(1) $f(u_1, \ldots, u_n) \sqcup f(v_1, \ldots, v_n) \rightsquigarrow f(u_1 \sqcup v_1, \ldots, u_n \sqcup v_n)$,
(2) $u \sqcup v \rightsquigarrow x_{u,v}$ otherwise, where $x_{u,v}$ is a variable uniquely determined by u and v.

[2]The author thanks Traian Florin Şerbănuţă for observing this.

Given terms s and t, the term obtained by applying this algorithm to $s \sqcup t$ is their anti-unification; the corresponding substitutions instantiating it to s and, respectively, t are obtained by instantiating each variable $x_{u,v}$ to u and, respectively, v. For example, $f(g(0),0) \sqcup f(g(1),1) \rightsquigarrow^* f(g(x_{0,1}),x_{0,1})$. Indeed, the term $f(g(x_{0,1}),x_{0,1})$ containing one variable, $x_{0,1}$, is the least general term that is more general than both $f(g(0),0)$ and $f(g(1),1)$. Also, the two original terms can be recovered by substituting $x_{0,1}$ with 0 and, respectively, 1.

Plotkin's algorithm can be mimicked with inference in matching logic. For the example above, the following matching logic proof blindly follows the application of Plotkin's algorithm (all proof steps correspond to applications of proof rules of FOL with equality):

$$\begin{aligned}
& f(g(0),0) \vee f(g(1),1) && = \\
& \exists x \,.\, \exists y \,.\, f(x,y) \wedge (x = g(0) \wedge y = 0 \vee x = g(1) \wedge y = 1) && = \\
& \exists x \,.\, \exists y \,.\, \exists z \,.\, f(x,y) \wedge x = g(z) \wedge (z = 0 \wedge y = 0 \vee z = 1 \wedge y = 1) && = \\
& \exists x \,.\, \exists y \,.\, \exists z \,.\, f(x,y) \wedge x = g(z) \wedge (z = 0 \wedge y = z \vee z = 1 \wedge y = z) && = \\
& \exists x \,.\, \exists y \,.\, \exists z \,.\, f(x,y) \wedge x = g(z) \wedge y = z \wedge (z = 0 \vee z = 1) && = \\
& \exists z \,.\, f(g(z),z) \wedge (z = 0 \vee z = 1)
\end{aligned}$$

The resulting pattern contains both the generalization, $f(g(z),z)$, and the two witness substitutions that can recover the original terms (encoded as a disjunction of equalities).

5.8. **Built-in Domains.** Dedicated solvers and decision procedures specialized for particular but important mathematical domains abound in the literature. Some domains may not even have finite descriptions in certain logics; a classic example is the domain of natural numbers, which does not admit a finite, not even a recursively enumerable axiomatization in FOL (Gödel's incompleteness [39, 40]). Therefore, to reason about certain properties that involve natural numbers, we need to leave FOL. The standard approach is to assume some oracle for the domain of interest, which is capable of answering validity questions within that domain. At the practical level, such an oracle may be implemented using specialized procedures and algorithms for that domain, such as those provided by Z3 [29], Yices [33], CVC [6, 5], etc. At the theoretical level, the set of models of the FOL specification in question is restricted to those that inherit the desired model for the built-in data-types, and thus we can assume all the valid FOL properties of that domain in the rest of the proof even if those properties are not provable using FOL.

Reasoning with built-in domains can be done exactly the same way in matching logic: assume the desired sorts and symbols for the built-in domains, together with all their valid patterns. Due to their ubiquitous nature, from now on we tacitly assume definitions of integers and of natural numbers, as well as of Boolean values, with common operations on them. We assume that these come with three sorts, *Int*, *Nat* and *Bool*, and the operations on them use the conventional syntax; e.g., $\_+\_ : Int \times Int \to Int$, $\_+\_ : Nat \times Nat \to Nat$, $\_ > \_ : Nat \times Nat \to Bool$, $\_ and \_ : Bool \times Bool \to Bool$, $not\_ : Bool \to Bool$, etc.

Boolean expressions are frequently used in matching logic specifications to constrain variables that occur in patterns of possibly other sorts. For example, suppose that in some domain *Real* of real numbers we want to refer to all numbers of the form $1/x$ where x is a strictly positive integer. These numbers are precisely matched by the pattern $1/x \wedge (x > 0 =^{Real}_{Bool} true)$. However, this pattern is too verbose. For the sake of a more compact and easy to read notation, we introduce the following:

Notation 5.27. *If b is a proper Boolean expression, that is, a term pattern of sort Bool (Definition 5.21), then we will write just b instead of $b = true$ in any other sort context.*

Notation 5.27 allows us to use Boolean expressions in any sort context, thanks to the additional notational conventions for equality in Section 5.2. For example, we can write $1/x \wedge x > 0$ instead of $1/x \wedge (x > 0 =^{Real}_{Bool} true)$.

To avoid confusion or even introducing inconsistencies, we urge the reader to respect the underlined words proper and other in Notation 5.27. That's because *Bool* expressions, when regarded as patterns, evaluate to one of the singleton sets $\{t\}$ (the true value) or $\{f\}$ (the false value), while patterns of sort *Bool* can evaluate to any of the four sets $\emptyset$, $\{t\}$, $\{f\}$, or $\{t, f\}$. For example, consider the *Bool* patterns $\top_{Bool}$ and $\bot_{Bool}$, which are not proper *Bool* expressions and evaluate to the sets $\{t, f\}$ and $\emptyset$, respectively, and an equality pattern $\top_{Bool} = \bot_{Bool}$ which is obviously $\bot$ (regardless of the sort context). If we carelessly apply the notation above to this pattern we get $(\top_{Bool} = true) = (\bot_{Bool} = true)$, that is, $\top$; that's because both $\top_{Bool} = true$ and $\bot_{Bool} = true$ are $\bot$, and $\bot = \bot$ is $\top$. So it is important to apply the notation "b as a shortcut for $b = true$" only when it is guaranteed that b evaluates to either $\{t\}$ or $\{f\}$, such as when b is a proper Boolean expression term. It is also important to apply the notation only when b occurs in sort contexts other than *Bool*. For example, consider the Boolean expression $b \equiv (true\ or\ false)$. If we carelessly apply the notation above to the Boolean sub-expressions *true* and *false*, which occur in b above in *Bool* contexts, then we get $(true = true)\ or\ (false = true)$, which is $\top_{Bool}\ or\ \bot_{Bool}$, which is $\bot_{Bool}$ (by the second item in Definition 2.3). On the other hand, $b = true$ is $\top_{Bool}$.

When reasoning with matching logic patterns, it is often the case that various Boolean expression constraints come from various sub-patterns. We would like to combine them into larger Boolean expressions, which we can then send to SMT solvers. The following result allows us to do that:

Proposition 5.28. *If b, b_1 and b_2 are proper Bool expressions, then*

- $\models (b_1 = true \wedge b_2 = true) = (b_1\ and\ b_2 = true)$
- $\models \neg(b = true) = (not\ b = true)$

Other similar properties for derived Boolean constructs can be derived from these.

Proof. Trivial: each of $\overline{\rho}(b)$, $\overline{\rho}(b_1)$, and $\overline{\rho}(b_2)$ can only be $\{t\}$ or $\{f\}$, for any valuation ρ. □

6. Instance: Algebraic Specifications and Beyond

An algebraic specification is a many-sorted signature (S, Σ) together with a set of equations[3] E over Σ-terms with variables. The variables are assumed universally quantified over the entire equation. The models, called Σ-algebras, are first-order Σ-models without predicates where equality is interpreted as the identity relation. We let $\models_{Alg}$ denote the algebraic specification satisfaction relation; in particular, $E \models_{Alg} e$ means that any model that satisfies all the equations in E also satisfies e.

Algebraic specifications play a crucial role in theoretical computer science and program reasoning, because they are often used to define abstract data types (ADTs) [60, 41]. Some common ADTs, which have proved useful in many applications, are lists (or sequences), sets, multisets, maps, multimaps, graphs, stacks, queues, priority queues, double-ended queues, double-ended priority queues, etc. [91]. These ADTs can be found a variety of formal definitions using algebraic specifications in the literature, not necessarily always equivalent, and are easily definable or already pre-defined in algebraic specification languages such as

[3]We only consider unconditional equations here.

Maude [25], CASL [63], CafeOBJ [31], OBJ [42], Clear [19], etc., as well as in many other languages with support for ADTs.

Here we show not only that algebraic specifications can be regarded as matching logic specifications, but also that the use of matching logic often allows for more expressive, concise and intuitive specifications of ADTs. To capture conventional ADTs as matching logic specifications, we need to do almost nothing besides recalling the conventions and notations introduced in Section 5. Specifically, algebraic equations $t = t'$ in E are regarded as matching logic equality patterns $t = t'$ (Section 5.2), algebraic symbols in Σ are interpreted as functional symbols (Section 5.4, Definition 5.21), and no other patterns but equations are allowed in specifications. The resulting matching logic specifications are then precisely the algebraic specifications not only syntactically, but also semantically:

Proposition 6.1. *Let (S, Σ, F) be the matching logic specification associated to the algebraic specification (S, Σ, E) as above, that is, F contains an equality pattern for each equation in E, as well as all the patterns stating that the symbols in Σ are interpreted as functions (see Definition 5.21). Then for any Σ-equation e, we have $E \models_{Alg} e$ iff $F \models e$.*

Proof. The key observation is that, in a similar style to the proof of Proposition 4.1, there is a bijection between the matching logic models M satisfying F and the (S, Σ)-algebras M' satisfying E, such that $M \models e$ iff $M' \models_{Alg} e$ for any Σ-equation e. The model bijection is defined as follows:

- $M'_s = M_s$ for each sort $s \in S$;
- $\sigma_{M'} : M_{s_1} \times \cdots \times M_{s_n} \to M_s$ with $\sigma_{M'}(a_1, \ldots, a_n) = a$ iff $\sigma_M : M_{s_1} \times \cdots \times M_{s_n} \to \mathcal{P}(M_s)$ with $\sigma_M(a_1, \ldots, a_n) = \{a\}$. Note that this is well-defined because F includes all the patterns stating that all the symbols are functional, so σ_M is a function.

This model relationship is easy to prove a bijection, and everything else follows from it. □

Using the notations introduced so far, Peano natural numbers, for example, can be defined as the following matching logic specification:

$$\begin{array}{ll} 0 : \to \mathit{PNat} & \\ \mathit{succ} : \mathit{PNat} \to \mathit{PNat} & \mathit{plus}(0, y) = y \\ \mathit{plus} : \mathit{PNat} \times \mathit{PNat} \to \mathit{PNat} & \mathit{plus}(\mathit{succ}(x), y) = \mathit{succ}(\mathit{plus}(x, y)) \end{array}$$

This looks identical to the conventional algebraic specification definition, which is precisely the point and justifies our notation conventions in Section 5. In particular, the functional notation (Definition 5.21) for the three symbols ensures that they will be interpreted as functions in the matching logic models. Also, as seen in the proof of Proposition 6.1, there is a bijective correspondence between the matching logic models of the specification above and the conventional models of the Peano algebraic specification (we only discuss the loose semantics here, not the initial-algebra semantics [44]).

Note, however, that matching logic allows more than just equational patterns. For example, we can add to F the pattern $0 \ \vee \ \exists x \,.\, \mathit{succ}(x)$ stating that any number is either 0 or the successor of another number. Nevertheless, since matching logic ultimately has the same expressive power as predicate logic (Proposition 10.1), we cannot finitely axiomatize in matching logic any mathematical domains that do not already admit finite FOL axiomatizations. As indicated in Section 5.8, we follow the standard approach to deal with built-in domains, namely we assume them theoretically presented with potentially infinitely many axioms but implemented using specialized decision procedures. Indeed, our matching logic implementation prototype in $\mathbb{K}$ defers the solving of all domain constraints to Z3 [29].

6.1. **Sequences, Multisets and Sets.** Sequences, multisets and sets are typical ADTs. Matching logic enables, however, some useful developments and shortcuts. For simplicity, we only discuss collections over *Nat*, and name the corresponding sorts *Seq*, *MultiSet*, and *Set*, respectively. Ideally, we would build upon an order-sorted algebraic signature setting, e.g. following the approach in [43], so that we can regard $x : Nat$ not only as an element of sort *Nat*, but also as one of sort *Seq* (a one-element sequence), as one of sort *MultiSet*, as well as one of sort *Set*. Extending matching logic to an order-sorted setting is beyond the scope of this paper. The reader who is not familiar with order-sorted concepts can safely assume that elements of sort *Nat* used in a *Seq*, *MultiSet*, or *Set* context are wrapped with injection symbols. The symbols below can have many equivalent definitions.

Sequences can be defined with two symbols and corresponding equations:

$$\begin{array}{ll} \epsilon : \to Seq & \epsilon \cdot x = x \\ \_ \cdot \_ : Seq \times Seq \to Seq & x \cdot \epsilon = x \\ & (x \cdot y) \cdot z = x \cdot (y \cdot z) \end{array}$$

We assume that lowercase variables have sort *Nat* and uppercase variables have the appropriate collection sort. To avoid adding initiality constraints on models, yet be able to do proofs by case analysis and elementwise equality, we can add the following patterns:

$$\begin{array}{l} \epsilon \vee \exists x \,.\, \exists S.\, x \cdot S \\ (x \cdot S = x' \cdot S') = (x = x') \wedge (S = S') \end{array}$$

The second axiom above holds strictly for sequences, but not for commutative collections, so it needs to be removed later when we add the commutativity axiom to define multisets and sets. We next define two common operations on sequences:

$$\begin{array}{ll} rev : Seq \to Seq & \_ \in \_ : Nat \times Seq \to Bool \\ rev(\epsilon) = \epsilon & x \in \epsilon = false \\ rev(x \cdot S) = rev(S) \cdot x & x \in y \cdot S = (x = y \wedge true) \ or \ x \in S \end{array}$$

To illustrate the flexibility of matching logic, we next define up-to and Fibonacci sequences of natural numbers.

$$upto : Nat \to Seq \qquad upto(n) = (n = 0 \wedge \epsilon \vee n > 0 \wedge upto(n-1) \cdot n)$$

This specification needs to be explained. Let M be a model satisfying the above. First recall Notation 5.27. For notational simplicity, assume that M_{Nat} and M_{Seq} are the sets of natural numbers and of comma-separated sequences of natural numbers, respectively. We show by induction on m that $upto_M(m) = \{1 \cdot 2 \cdots m\}$. If $m = 0$ then the second disjunct of the axiom is $\emptyset$ and thus the first disjunct ensures that $upto_M(0) = \epsilon_M = \epsilon$. If $m > 0$ then the first disjunct is $\emptyset$ and thus the second disjunct, with $\rho : Var \to M$ such that $\rho(n) = m$, yields $\overline{\rho}(upto(n)) = \overline{\rho}(upto(n-1) \cdot n)$, that is, $upto_M(m) = upto_M(m-1) \cdot m$, which by the induction hypothesis implies $upto_M(m) = \{1 \cdot 2 \cdots m-1\} \cdot m = \{1 \cdot 2 \cdots m\}$.

Similarly, we can specify a function that defines the sequence of Fibonacci numbers up to a given number $n > 0$:

$$\begin{array}{ll} fib : Nat \to Seq & fib(n) = (n = 0 \wedge 0 \vee n > 0 \wedge aux(n,\ 0 \cdot 1)) \\ aux : Nat \times Seq \rightharpoonup Seq & aux(1,\ S) = S \\ & n > 1 \to aux(n,\ S \cdot x \cdot y) = aux(n-1,\ S \cdot x \cdot y \cdot x + y) \end{array}$$

We conclude the discussion on sequences with an elegant means to sort sequences following a bubble-sort methodology:

$$(x \cdot y \wedge x > y) = y \cdot x$$

Since equations are symmetric, the above effectively allows to prove (so far only semantically, in a model) a sequence equal to any of its permutations, i.e., sequences become multisets. If the equations were oriented, like they are in reachability logic [80], then the above would yield a sequence sorting procedure.

We can transform sequences into multisets adding the equality axiom $x \cdot y = y \cdot x$, and into sets by also including $x \cdot x = \bot$ or $x \cdot x = x$. Here is one way to axiomatize intersection:

$$\_ \cap \_ : \mathit{Set} \times \mathit{Set} \to \mathit{Set} \qquad \begin{array}{l} \epsilon \cap S_2 = \epsilon \\ (x \cdot S_1) \cap S_2 = ((x \in S_2 \to x) \wedge (\neg(x \in S_2) \to \epsilon)) \cdot (S_1 \cap S_2) \end{array}$$

6.2. **Maps.** Finite-domain maps are also a typical ADT. Due to their ubiquity in defining algebraic specifications, maps are usually predefined in languages like those mentioned in the preamble of this section (Section 6). For example, below is a snippet of Maude's `MAP` module [25], which is parametric in both the source and the target domains:

```
fmod MAP{X :: TRIV, Y :: TRIV} is
  sorts Entry{X,Y} Map{X,Y} .
  subsorts Entry{X,Y} < Map{X,Y} .
  op _|->_ : X$Elt Y$Elt -> Entry{X,Y} [ctor] .
  op empty : -> Map{X,Y} [ctor] .
  op _,_ : Map{X,Y} Map{X,Y} -> Map{X,Y} [assoc comm id: empty ctor] .
...
endfm
```

Note that the map concatenation symbol, "`_,_`", is associative (attribute "`assoc`"), commutative ("`comm`"), and has the "`empty`" map as identity ("`id: empty`"). The attribute "`ctor`" states that the corresponding symbols are constructors. Additional axioms, not shown here, ensure that maps are always well-formed, that is, maps with multiple bindings of the same key are disallowed. When instantiated with natural numbers for both the domain and the target, this `MAP` module defines well-formed finite-domain maps such as "`1 |-> 5, 2 |-> 0, 7 |-> 9, 8 |-> 1`". In Section 9, to show how separation logic can be framed as a matching logic theory with essentially zero representational/encoding distance, we will pick an instance of maps with natural numbers as both the domain and the co-domain, and will rename `empty` to *emp* and `_,_` to $\_ * \_$.

7. Instance: First-Order Logic

First-order logic (FOL) extends predicate logic with function symbols and allows predicates to apply to terms with variables built using the function symbols. Recall from Section 4 that by "predicate logic" in this paper we mean what is also called "pure predicate logic" in the literature, namely FOL without any function or constant symbols.

Formally, given a FOL signature (S, Σ, Π), the syntax of its (many-sorted) formulae is:

$$\begin{array}{rcll} t_s & ::= & x \in \mathit{Var}_s & \\ & | & \sigma(t_{s_1}, \ldots, t_{s_n}) & \text{with } \sigma \in \Sigma_{s_1 \ldots s_n, s} \\ \varphi & ::= & \pi(t_{s_1}, \ldots, t_{s_n}) & \text{with } \pi \in \Pi_{s_1 \ldots s_n} \\ & | & \neg\varphi & \\ & | & \varphi \wedge \varphi & \\ & | & \exists x.\varphi & \end{array}$$

Compare the above with the syntax of matching logic in Section 2. Unlike FOL, matching logic does not distinguish between the term and predicate syntactic categories, reason for which its syntax is in fact more compact than FOL's. Moreover, matching logic allows logical constructs over all the syntactic categories, not only over predicates, and locally where they are needed instead of only at the top, predicate level. Also, matching logic allows quantification over any sorts, including over sorts of symbols thought of as predicates.

Like with predicate logic (Section 4), we can instantiate matching logic to capture FOL as is, modulo the notational conventions in Section 5 but without any translations from one logic to the other. Like in predicate logic, we add a *Pred* sort and regard the FOL predicate symbols as matching logic symbols of result *Pred*, and disallow variables of sort *Pred* and restrict the use of logical connectives and quantifiers to only patterns of sort *Pred*. Then there is a one-to-one correspondence between FOL formulae and matching logic patterns of sort *Pred*; we let φ range over them. Moreover, following the approach in Section 5.4, we constrain each FOL operational symbol $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ to be interpreted as a function, that is, with the notation introduced in Section 5.4, we write the symbols meant to be functions as $\sigma : s_1 \ldots s_n \to s$. Formally, let (S^{ML}, Σ^{ML}) be the matching logic signature with $S^{ML} = S \cup \{Pred\}$ and $\Sigma^{ML} = \Sigma \cup \{\pi : s_1 \ldots s_n \to Pred \mid \pi \in \Pi_{s_1 \ldots s_n}\}$, and let F be $\{\exists z{:}s \,.\, \sigma(x_1{:}s_1, \ldots, x_n{:}s_n) = z \mid \sigma \in \Sigma_{s_1 \ldots s_n, s}\}$ stating that each symbol in Σ is a function.

Proposition 7.1. *For any FOL formula φ, we have $\models_{FOL} \varphi$ iff $F \models \varphi$.*

Proof. The proof is similar to that of Proposition 4.1. Like there, the implication "$\models_{FOL} \varphi$ implies $F \models \varphi$" follows by the completeness of FOL. Indeed, it is well-known that the properties in Proposition 2.8 and Corollary 5.23, when regarded as proof rules for deriving FOL formulae φ, yield a sound and complete proof system for FOL [39]. That is, "$\models_{FOL} \varphi$ iff $\vdash_{FOL} \varphi$". However, since Proposition 2.8 and Corollary 5.23 show that these proof rules are also sound for matching logic in the context of F (Corollary 5.23 requires that t is a term pattern in the substitution rule, which holds in the context of F), we conclude that "$\vdash_{FOL} \varphi$ implies $F \models \varphi$". Therefore, "$\models_{FOL} \varphi$ implies $F \models \varphi$".

For the other implication, we also follow the idea in Proposition 4.1. From any FOL model $M^{FOL} = (\{M^{FOL}_s\}_{s \in S}, \{\sigma_{M^{FOL}}\}_{\sigma \in \Sigma}, \{\pi_{M^{PL}}\}_{\pi \in \Pi})$ we can construct a matching logic model $M^{ML} = (\{M^{ML}_s\}_{s \in S \cup \{Pred\}}, \{\sigma_{M^{ML}}\}_{\sigma \in \Sigma} \cup \{\pi_{M^{ML}}\}_{\pi \in \Pi})$, where $M^{ML}_s = M^{FOL}_s$ for all sorts $s \in S$ and $M^{ML}_{Pred} = \{\star\}$ (with $\star$ some arbitrary but fixed element), $\sigma_{M^{ML}}(a_1, \ldots, a_n) = \{\sigma_{M^{FOL}}(a_1, \ldots, a_n)\}$, and $\pi_{M^{ML}}(a_1, \ldots, a_n) = \{\star\}$ iff $\pi_{M^{PL}}(a_1, \ldots, a_n)$ holds, and otherwise $\pi_{M^{ML}}(a_1, \ldots, a_n) = \emptyset$. Note that $M^{ML} \models F$: indeed, $\sigma_{M^{ML}}(a_1, \ldots, a_n)$ contains precisely one element for any $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ and any $a_1 \in M^{ML}_{s_1}$, ..., $a_n \in M^{ML}_{s_n}$, namely $\sigma_{M^{FOL}}(a_1, \ldots, a_n)$. It therefore suffices to show, for any FOL formula φ, that $M^{FOL} \models_{FOL} \varphi$ iff $M^{ML} \models_{ML} \varphi$. Like in Proposition 4.1, we can show by structural induction on φ that for any $\rho : Var \to M^{FOL}$, it is the case that $M^{FOL}, \rho \models_{FOL} \varphi$ iff $\overline{\rho}(\varphi) = \{\star\}$. The induction proof differs from that in Proposition 4.1 only in the base case, where we need to notice that term patterns are functional in M^{ML}, thanks to Corollary 5.22, and that $\overline{\rho}(t) = a$ for a term t in the FOL context (with $\rho : Var \to M^{FOL}$) iff $\overline{\rho}(t) = \{a\}$ in the matching logic context (where ρ is regarded as $\rho : Var \to M^{ML}$, which is possible as we disallow *Pred* variables). □

Consequently, FOL is also a methodological fragment of matching logic. Moreover, since the rules of FOL where we replace all the predicate meta-variables with pattern meta-variables are sound for matching logic, we can use off-the-shelf decision procedures and solvers for FOL or fragments of it *unchanged* when doing matching logic reasoning. The only thing we have to be careful about is to distinguish the term patterns from arbitrary

patterns, and make sure that the Substitution rule of FOL is only applied with t a term pattern, otherwise it may be unsound. Section 11 discusses this in detail.

Predicate logic and FOL with equality also fall as methodological fragments of matching logic. In addition to the constructions in Section 4 and, respectively, above in this section, all we have to do is to is to make use of the definedness symbols that we assume by convention included in all signatures (Section 5.1), which give us equality as an alias as described in Section 5.2. We leave the details as an exercise for the interested reader.

Like Boolean expressions, FOL is also frequently used in matching logic specifications to constrain variables that occur in patterns of possibly other sorts. Consider the same example we discussed before and after Notation 5.27, where we want to refer to all real numbers of the form $1/x$ with x a strictly positive integer, but this time using a given predicate $\mathit{positive}?(x)$ that tells whether x is positive. We can use the pattern $1/x \wedge (\mathit{positive}?(x) =^{\mathit{Real}}_{\mathit{Pred}} \top_{\mathit{Pred}})$, but that is too verbose. We would like to just write $1/x \wedge \mathit{positive}?(x)$. Following Notation 5.27, we introduce the following similar notation for predicates instead of Boolean expressions:

Notation 7.2. *If φ is a FOL formula, we take the freedom to write φ instead of $\varphi = \top_{\mathit{Pred}}$.*

Since both the FOL formulae and the patterns of *Pred* sort evaluate to only two possible values, $\emptyset$ or $\{\star\}$, unlike Notation 5.27 we can freely apply the notation above in any contexts, including of sort *Pred*. Note, however, that care must be exercised to ensure that φ is indeed a FOL formula. For example, if one extends FOL with additional formula constructs, like separation logic does for example (Section 9), then the above notation may lead to inconsistencies. As discussed in Section 9 in detail, matching logic has a different way to deal with such extensions (allowing different sorts of "predicates"), without polluting the universe of FOL formulae and thus allowing the notation above to still apply.

Same as with Boolean expressions in Proposition 5.28, we sometimes need to combine various FOL constraints resulting from various sub-patterns in order to make appropriate calls to FOL provers, e.g., SMT solvers. The following result allows us to do that:

Proposition 7.3. *If p, p_1 and p_2 are FOL formulae, then*

- $\models (p_1 = \top_{\mathit{Pred}} \wedge p_2 = \top_{\mathit{Pred}}) = (p_1 \wedge p_2 = \top_{\mathit{Pred}})$
- $\models \neg(p = \top_{\mathit{Pred}}) = (\neg p = \top_{\mathit{Pred}})$

Other similar properties for derived FOL constructs can be derived from these.

Proof. Trivial: each of $\overline{\rho}(p)$, $\overline{\rho}(p_1)$, and $\overline{\rho}(p_2)$ can only be $\emptyset$ or $\{\star\}$, for any valuation ρ. □

8. Instance: Modal Logic

It turns out that the vanilla matching logic over just one sort with (countably many) constants and definedness (as defined in Section 5.1) captures one of the most popular modal logics, S5 [7, 57, 45]. At the end of this section we briefly discuss how other modal logics can also be framed as matching logic instances, but until there we only discuss S5 and thus take the liberty to implicitly mean the "S5 modal logic" whenever we say "modal logic".

We start by giving the syntax and semantics of modal logic. Let $\mathit{Var}_{\mathit{Prop}}$ be a countable set of *propositional variables* p, q, etc. Then the modal logic syntax is defined as follows:

$$\begin{array}{rcl} \varphi & ::= & \mathit{Var}_{\mathit{Prop}} \\ & | & \neg\varphi \\ & | & \varphi \rightarrow \varphi \\ & | & \Box\varphi \end{array}$$

The remaining propositional constructs $\wedge$, $\vee$ and $\leftrightarrow$, can be defined as derived constructs. Therefore, syntactically, modal logic adds the $\Box$ construct to propositional logic, which is called *necessity*: $\Box\varphi$ is read "it is necessary that φ". The dual *possibility* construct can be defined as a derived construct: $\Diamond\varphi \equiv \neg\Box\neg\varphi$ is read "it is possible that φ". Semantically, the truth value of a formula is relative to a "world". Propositions can be true in some worlds but false in others, and thus formulae can also be true in some worlds but not in others:

Definition 8.1. *Let W be a set of **worlds**. Mappings $v : Var_{Prop} \times W \to \{true, false\}$, called **(modal logic) W-valuations**, state that each proposition only holds in a given (possibly empty or total) subset of worlds. Valuations extend to modal logic formulae:*

- $v(\neg\varphi, w) = true$ *iff* $v(\varphi, w) = false$
- $v(\varphi_1 \to \varphi_2, w) = true$ *iff* $v(\varphi_1, w) = false$ *or* $v(\varphi_2, w) = true$
- $v(\Box\varphi, w) = true$ *iff* $v(\varphi, w') = true$ *for every* $w' \in W$

*Formula φ is **valid in** W, written $W \models_{\text{S5}} \varphi$, iff $v(\varphi, w) = true$ for any W-valuation v and any $w \in W$. Formula φ is **valid**, written $\models_{\text{S5}} \varphi$, iff $W \models_{\text{S5}} \varphi$ for all W.*

Modal logic (S5) admits the following sound and complete proof system [7, 57]:

(N) Rule: If φ derivable then $\Box\varphi$ derivable
(K) Axiom: $\Box(\varphi_1 \to \varphi_2) \to (\Box\varphi_1 \to \Box\varphi_2)$
(M) Axiom: $\Box\varphi \to \varphi$
(5) Axiom: $\Diamond\varphi \to \Box\Diamond\varphi$

We next show that we can define a matching logic specification (S, Σ, F) which faithfully captures modal logic, both syntactically and semantically. The idea is quite simple: S contains precisely one sort, say $World$; Σ contains one constant symbol $p \in \Sigma_{\lambda, World}$ for each propositional variable $p \in Var_{Prop}$, plus a unary symbol $\Diamond \in \Sigma_{World, World}$; and F contains precisely one axiom stating that $\Diamond$ is the definedness symbol (Section 5.1), namely $\Diamond x : World$ (x is a free $World$ variable in this pattern). Then we let $\Box\varphi$ be the totality construct (Notation 5.3), that is, syntactic sugar for $\neg\Diamond\neg\varphi$. Note that any modal logic formula φ can be regarded, as is, as a ground matching logic pattern over this signature; by "ground" we mean a pattern without variables, so the other implication is also true, because disallowing variables includes disallowing quantifiers. Moreover, Corollary 5.5 implies that the modal logic proof system above is sound for the resulting matching logic specification, so $\models_{\text{S5}} \varphi$ implies $\models \varphi$. We show the stronger result that the world/valuation models of modal logic are essentially identical the the matching logic (S, Σ, F)-models, and thus:

Proposition 8.2. *For any modal logic formula φ, we have $\models_{\text{S5}} \varphi$ iff $\models \varphi$.*

Proof. For any world W and W-valuation $v : Var_{Prop} \times W \to \{true, false\}$ (Definition 8.1), let $M_{W,v}$ be the matching logic (S, Σ, F)-model whose carrier is W, whose constant symbols $p \in \Sigma_{\lambda, World}$ (i.e., $p \in Var_{Prop}$) are interpreted as the sets of worlds $p_{M_{W,v}} = \{w \in W \mid v(p, w) = true\}$, and $\Diamond w$ is the total set W for each $w \in W$. Similarly, for each matching (S, Σ, F)-model M let $W_M = M_{World}$ be its carrier and let $v_M : Var_{Prop} \times W_M \to \{true, false\}$ be defined as $v(p, w) = true$ iff $w \in p_M$. It is clear that the two mappings defined above, $(W, v) \mapsto M_{W,v}$ and respectively $M \mapsto (W_M, v_M)$, are inverse to each other.

Since a modal logic formula φ can be regarded as a matching logic pattern with no variables, $\overline{\rho}(\varphi)$ only depends on the model M but not on any particular valuation $\rho : Var \to M$ (by (1) in Proposition 2.6). Let us then use the notation φ_M for the (unique) interpretation of φ in matching logic model M; note that $\varphi_M \subseteq M_{World}$.

We show that for any W and any W-valuation v, we have $v(\varphi, w) = \mathit{true}$ iff $w \in \varphi_{M_{W,v}}$. We show it by structural induction on φ. The cases when φ is a propositional symbol or a logical connective are trivial. For the necessity modal construct $\Box$, we have $v(\Box\varphi, w) = \mathit{true}$ iff $v(\varphi, w') = \mathit{true}$ for all $w' \in W$ (Definition 8.1), iff $w' \in \varphi_{M_{W,v}}$ for all $w' \in W$ (induction hypothesis), iff $\varphi_{M_{W,v}} = W = (M_{W,v})_{\mathit{World}}$, iff $(\Box\varphi)_{M_{W,v}} = W$ (by Proposition 5.4), iff $w \in (\Box\varphi)_{M_{W,v}}$ (by Proposition 5.4). Therefore, $v(\varphi, w) = \mathit{true}$ iff $w \in \varphi_{M_{W,v}}$.

We are now ready to prove the main result: $\models_{\mathrm{S5}} \varphi$ iff $v(\varphi, w) = \mathit{true}$ for any W and W-valuation v and $w \in W$ (Definition 8.1), iff $w \in \varphi_{M_{W,v}}$ for any W and W-valuation v and $w \in W$ (by the property proved above by structural induction), iff $\varphi_{M_{W,v}} = W$ for any W and W-valuation v, iff $M_{W,v} \models \varphi$ for any W and W-valuation v (by Proposition 2.5), iff $M \models \varphi$ for any matching logic (S, Σ, F)-model M (because of the bijective correspondence between pairs (W, v) and (S, Σ, F)-models M proved above), iff $\models \varphi$. □

The result above, together with the general translation of matching logic to predicate logic with equality discussed in Section 10, will also give us a translation of modal logic to predicate logic with equality. Translations from modal logic to various types of first-order (or second-order or other even more expressive) logics are well-known in the literature, one of them to predicate logic being called the "standard translation" [90, 11]. Our goal in this section was not to propose yet another translation, but to show how modal logic can be framed as a matching logic specification *without any translation*.

There are many variants of modal logic [90, 11, 45]. One may naturally wonder if all of them can be similarly regarded as matching logic theories. While systematically investigating each and everyone of them seems tedious and likely not worth the effort, it is nevertheless interesting to note that there is an immediate connection between one of the most general variants of modal logic, called *multi-dimensional* or *polyadic modal logic* [11], and matching logic. Instead of particular unary modal operators like $\Box$ and $\Diamond$, polyadic modal logic allows arbitrary operators taking any number of formula arguments; if Δ is such an operator of n arguments and φ_1, ..., φ_n are formulae, then $\Delta(\varphi_1, ..., \varphi_n)$ is also a formula. In models, called *general frames*, each such operator Δ is associated a relation R_Δ of $n+1$ arguments. Propositional variables are also interpreted as sets (of "worlds in which they hold") by valuations, and given set of worlds W, valuation $v : \mathit{Var}_{\mathit{Prop}} \times W \to \{\mathit{true}, \mathit{false}\}$ and world $w \in W$, we have $v(\Delta(\varphi_1, ..., \varphi_n), w) = \mathit{true}$ iff there are $w_1, ..., w_n \in W$ such that $v(\varphi_1, w_1) = \mathit{true}$, ..., $v(\varphi_n, w_n) = \mathit{true}$ and $R_\Delta(w, w_1, ..., w_n)$.

It is easy to associate a matching logic specification (S, Σ, F) to any polyadic modal logic. Like for $S5$, we let S contain precisely one sort, World, and Σ contain one constant symbol $p \in \Sigma_{\lambda, \mathit{World}}$ for each propositional variable $p \in \mathit{Var}_{\mathit{Prop}}$. Further, we add a symbol $\Delta \in \Sigma_{\mathit{World} \times \cdots \times \mathit{World}, \mathit{World}}$ of n arguments for each polyadic modal operator Δ taking n arguments. Then any polyadic modal logic formula φ can be regarded without any change/-translation as a matching logic formula. Further, any axioms/schemas in polyadic modal logic can be added as matching logic axioms/schemas in F. Then we can extend Proposition 8.2 to show $\models \varphi$ in the polyadic modal logic iff $F \models \varphi$ in matching logic; the key technical insight here is that there is a bijective correspondence between relations of $n+1$ arguments and functions of n arguments returning sets of elements.

When compared to polyadic modal logic, matching logic has a couple of advantages which, in our view, make it more appealing to use in practice. First, it has sorts. Thus, unlike polyadic modal logic which only has "formulae", matching logic allows us to have patterns of various types. For example, in Section 2.2 we show how heap patterns interact

with program patterns and how all can be put together in configuration patterns; while possible in theory, it would be quite inconvenient to force all patterns to have the same sort. Second and more importantly, modal logic and matching logic have a different interpretation for "variables". In modal logic (propositional) variables are interpreted as sets and we are not allowed to quantify over them, while in matching logic variables are interpreted as just elements and we can quantify over them. Like shown above, the set interpretation can be recovered in matching logic by associating constant symbols to propositional variables. But the singleton interpretation of variables in matching logic, combined with the capability to quantify over variables of any sort, allows us to elegantly define many useful properties, such as those in Section 5. For example, the simple pattern $\forall x \,.\, \lceil x \rceil$ defines the semantics of the definedness symbol $\lceil \_ \rceil$, which as seen above gives us the $\Diamond$ construct of S5. It is critical that x ranges over singleton elements in models. If one attempts to do the same in polyadic modal logic naively replacing x with a propositional variable p, then one gets an inconsistent theory (because we want $\lceil p \rceil$ to be false when p is interpreted as the empty set of worlds). Definedness then allows us to define membership and equality, and thus allows us to use patterns like $\forall x \,.\, \exists y \,.\, f(x) = y$ to state that symbol f is a function, etc.

Whether the results and observations above have practical relevance remains to be seen. We hope they at least enhance our understanding of both matching logic and modal logic.

9. Instance: Separation Logic

Matching logic has inherent support for structural separation, without a need for any special logic constructs or extensions. Indeed, pattern matching has a spatial meaning by its very nature: matching a subterm already separates that subterm from the rest of the context, so matching two or more terms can only happen when there is no overlapping between them. Moreover, matching logic patterns can combine structure with logical constraints, which allows not only to define very sophisticated patterns, but also to reason about patterns as if they were logical formulae, and to achieve modularity by globalizing local reasoning. Finally, since matching logic allows variables of any sorts, including of sort *Map* when heaps are concerned, it has inherent support for heap framing and local reasoning, too.

9.1. **Separation Logic Basics.** Separation logic (originating with ideas in [69, 68], followed by canonical work in [78], with more recent developments in [73, 18, 24, 17, 55] and with several provers supporting it in [8, 2, 15, 64, 9, 65, 72, 75, 73]), is a logic specifically crafted for reasoning about heap structures. There are many variants, but here we only consider the original variant in [68, 78]. Moreover, here we only discuss separation logic as an assertion-language, used for specifying state properties, and not its extension as an axiomatic programming language semantic framework. We regard the latter as an orthogonal aspect, which can similarly be approached using matching logic.

Separation logic extends the syntax of formulae in FOL (Section 7) as follows:

$$\begin{array}{rcll} \varphi & ::= & \textit{(FOL syntax)} & \\ & \mid & \mathit{emp} & \\ & \mid & \mathit{Nat} \mapsto \mathit{Nat} & \\ & \mid & \varphi * \varphi & \\ & \mid & \varphi \mathrel{-\!\!*} \varphi & \text{“magic wand”} \end{array}$$

Its semantics is based on a fixed model of stores and heaps, which are finite-domain maps from variables and, respectively, locations (which are particular numbers), to integers. Below we recall the formal definition of satisfaction in the original variant of separation logic, noting that subsequent variants of separation logic extend the underlying model to include stacks (instead of stores) as well as various types of resources that are encountered in modern programming languages. Such extensions are ignored here because they would only complicate the presentation without changing the overall message: they would only add more symbols to the corresponding matching logic signature with appropriate interpretations in the underlying model, and Theorem 9.2 would still hold. Nevertheless, we leave the thorough analysis of the diversity of separation logic variants proposed in the last 15 years through the lenses of matching logic as a subject for future work.

Definition 9.1. *(Separation logic semantics, adapted from* [68, 78]*) Partial finite-domain maps $s : Var \rightharpoonup Nat$ are called **stores**, partial finite-domain maps $h : Nat \rightharpoonup Nat$ are called **heaps**, and pairs (s, h) of a store and a heap are called **states**. The semantics of the separation logic constructs are given in terms of such states, as follows:*

- *$(s, h) \models_{SL} \varphi$ for a FOL formula φ iff $s \models_{FOL} \varphi$ (the heap portion of the model is irrelevant for the FOL fragment);*
- *$(s, h) \models_{SL} emp$ iff $Dom(h) = \emptyset$;*
- *$(s, h) \models_{SL} e_1 \mapsto e_2$ where e_1 and e_2 are terms of sort Nat (thought of as "expressions") iff $Dom(h) = \overline{s}(e_1) \neq 0$ and $h(\overline{s}(e_1)) = \overline{s}(e_2)$, where $\overline{s}$ is the (partial function) extension of s to expressions (with variables) of sort Nat, defined similarly to the extension of valuations to patterns in Definition 2.3;*
- *$(s, h) \models_{SL} \varphi_1 * \varphi_2$ iff there exist h_1 and h_2 such that $Dom(h_1) \cap Dom(h_2) = \emptyset$ and $h = h_1 * h_2$ (the merge of h_1 and h_2, a partial function on maps written as an associative/commutative comma in Section 6.2) and $(s, h_1) \models_{SL} \varphi_1$, $(s, h_2) \models_{SL} \varphi_2$;*
- *$(s, h) \models_{SL} \varphi_1 -\!\!* \varphi_2$ iff for any h_1 with $Dom(h_1) \cap Dom(h) = \emptyset$, if $(s, h_1) \models_{SL} \varphi_1$ then $(s, h * h_1) \models_{SL} \varphi_2$; i.e., the semantics of "magic wand" is defined as the states whose heaps extended with a fragment satisfying φ_1 result in ones satisfying φ_2.*

*Separation logic formula φ is **valid**, written $\models_{SL} \varphi$, iff $(s, h) \models_{SL} \varphi$ for any state (s, h).*

9.2. **Map Patterns.** One of the most appealing aspects of separation logic is that it allows us to define compact and elegant specifications of heap abstractions using inductively defined predicates. Such an abstraction which is quite common is the linked-list abstraction $list(x, S)$ stating that x points to a linked list containing an abstract sequence of natural numbers S:

$$\begin{array}{rcl} list(x, \epsilon) & \stackrel{def}{=} & emp \wedge x = 0 \\ list(x, n \cdot S) & \stackrel{def}{=} & \exists z \,.\, x \mapsto [n, z] * list(z, S) \end{array}$$

Above, ϵ is the empty sequence, $n \cdot S$ is the sequence starting with natural number n and followed by sequence S, and $x \mapsto [n, z]$ is syntactic sugar for $x \mapsto n * (x + 1) \mapsto z$. So a linked list starting with address x takes either empty heap space, in which case x must be 0 and the abstracted sequence is ϵ, or there is some node in the linked list at location x in the heap that holds the head of the abstracted sequence (n) and a link (z) to another linked list that holds the tail of the abstracted sequence (S). The implicit properties of the implicit map model (the heap) in Definition 9.1 ensures the well-definedness of this abstraction, which essentially means that all the locations reached by unfolding a list abstraction using

the inductive properties above are distinct. The symbol $\stackrel{def}{=}$, sometimes written $\equiv$ in the literature, is not part of separation logic; it is a meta-logical means to define inductive, or recursive predicates, also encountered in the context of first-order logic: the predicate in question is interpreted in models as the least-fixed point of its defining (meta-)equations.

We next show that similar heap patterns can be defined directly in matching logic. Specifically, we pick a particular signature (for maps/heaps) together with desired axioms, that is, a matching logic specification, and show how additional patterns can be defined in the context of that specification. The definitions are as compact and elegant as those in separation logic, and no meta-logical constructs, such as $\stackrel{def}{=}$ or $\equiv$, appear to be necessary.

In what follows, we only discuss maps from natural numbers to natural numbers, but they can be similarly defined over arbitrary domains as keys and as values. Consider a matching logic specification of maps like the one shown in Section 6.2, but instantiated to natural numbers as both keys and values, with its axioms explicitly listed, and with a syntax that deliberately resembles that of separation logic (i.e., we use "*" instead of ","):

$$\begin{array}{ll} \_ \mapsto \_ : \mathit{Nat} \times \mathit{Nat} \rightharpoonup \mathit{Map} & \mathit{emp} * H = H \\ \mathit{emp} : \rightarrow \mathit{Map} & H_1 * H_2 = H_2 * H_1 \\ \_ * \_ : \mathit{Map} \times \mathit{Map} \rightharpoonup \mathit{Map} & (H_1 * H_2) * H_3 = H_1 * (H_2 * H_3) \\ 0 \mapsto a = \bot & x \mapsto a * x \mapsto b = \bot \end{array}$$

Recall that there are no predicates here, only patterns. When regarding the above ADT as a matching logic specification, we can prove that the bottom two pattern equations above are equivalent to $\neg(0 \mapsto a)$ and, respectively, $(x \mapsto a * y \mapsto b) \rightarrow x \neq y$, giving the $\_ \mapsto \_$ and $\_ * \_$ the feel of "predicates". Maps, like natural numbers, do not admit finite (or even recursively enumberable) equational (or first-order) axiomatizations, so adding a "good enough" subset of equations is the best we can do in practice. We chose ones that have been proposed by algebraic specification languages and by separation logics for several reasons. First, they have been extensively used, so there is a good chance they are "good enough" for many purposes. Second, we do not want to imply that we propose a novel axiomatization of maps; our novelty is the presentation of known specifications of maps using the general infrastructure of matching logic at no additional translation cost, without a need to craft a new logic to address the particularities of maps. Third, this will ease our presentation in Section 9.3 where the connection with such a logic specifically crafted for maps is discussed.

Consider the canonical model of partial maps M, where: $M_{Nat} = \{0, 1, 2, \ldots\}$; $M_{Map} =$ partial maps from natural numbers to natural numbers with finite domains and undefined in 0, with *emp* interpreted as the map undefined everywhere, with $\_ \mapsto \_$ interpreted as the corresponding one-element partial map except when the first argument is 0 in which case it is undefined (note that $\_ \mapsto \_$ was declared using $\rightharpoonup$), and with $\_ * \_$ interpreted as map merge when the two maps have disjoint domains, or undefined otherwise (note that $\_ * \_$ was also declared using $\rightharpoonup$). M satisfies all axioms above.

Following similar definitions in the context of separation logic, we next define several patterns that are commonly used when proving properties about programs that can allocate and de-allocate data-structures in the heap. We emphasize that our matching logic specifications below look almost identical, if not identical, to their separation logic variants. Which is, in fact, the main point we are making in this subsection. That is, that matching logic allows us to specify the same complex heap predicates as separation logic, equally compactly and elegantly, but without a need to devise any new heap-specific logic for that.

We start with matching logic definitions for complete linked lists and for list fragments. Let $list \in \Sigma_{Nat,Map}$ and $lseg \in \Sigma_{Nat \times Nat,Map}$ be two symbols together with patterns

$$list(0) = emp \qquad lseg(x,x) = emp$$
$$list(x) \wedge x \neq 0 = \exists z \,.\, x \mapsto z * list(z) \qquad lseg(x,y) \wedge x \neq y = \exists z \,.\, x \mapsto z * lseg(z,y)$$

Note that *list* and *lseg* are not meant to be functions, so we did not use the functional notation (Section 5.4) for them. Moreover, note that *lseg* is not even meant to be a totally defined relation (Section 5.6); indeed, $lseg(0,m)$ is $\emptyset$ (and not emp) for all $m > 0$.

The main difference between our definitions above and their separation logic variants is that the latter cannot use the (FOL) equality symbol as we did. Indeed, *list* and *lseg* are predicates there, same as equality, and predicates cannot take predicates as arguments. To define predicates like *list* and *lseg*, as seen at the beginning of this section, we have to *explicitly* use the meta-logical notation $\stackrel{\text{def}}{=}$ or $\equiv$ for defining "recursive predicates": predicates satisfying desired properties which have a least fixed-point interpretation in models. We emphasized "explicitly" above to distinguish it from the *implicit* least fixed-point principles used when mathematically defining the semantics of any logic. For example, in our context, the extension of ρ to $\overline{\rho}$ in Definition 2.3 uses a least-fixed point construction, similar to any other logic with terms, but that is orthogonal to how symbols are interpreted in the given model (symbol interpretation is given by the model, not by ρ).

There are two important questions about the matching logic specification above:

(1) Does this specification admit any solution in M, i.e., total relations $list_M : M_{Nat} \to \mathcal{P}(M_{Map})$ and $lseg_M : M_{Nat} \times M_{Nat} \to \mathcal{P}(M_{Map})$ satisfying the patterns above?
(2) If yes, is the solution unique? This is particularly important because we do not require initiality constraints on M nor smallest fixed-point constraints on solutions.

We answer these questions positively. We only discuss $lseg_M$, because the other is similar and simpler. A solution $lseg_M : M_{Nat} \times M_{Nat} \to \mathcal{P}(M_{Map})$ exists iff it satisfies the two pattern axioms for *lseg* above; explicitly, that means that any solution must satisfy:

$$lseg_M(n,n) = \{emp_M\} \text{ for all } n \geq 0$$
$$lseg_M(0,m) = \emptyset \text{ for all } m \neq 0$$
$$lseg_M(n,m) = \bigcup\{(\{n \mapsto_M n_1\} *_M lseg_M(n_1,m)) \mid n_1 \geq 0\} \text{ for all } n \neq 0 \text{ and } n \neq m$$

where $\_ *_M \_$ is M's merge function explained above extended to sets of maps for each argument; recall that the map merge function is undefined (i.e., it yields an empty set of maps) when the two argument maps are not merge-able. Note that we had to split the interpretation of the second equation pattern for *lseg* into two equalities, reflecting a case analysis on whether the first argument is 0 or not. Note also that $lseg(n,m) \neq \emptyset$ when $n \neq 0$, and that $lseg(n,m)$ contains only non-empty maps when $n \neq 0$ and $n \neq m$.

First, we claim that the following is a solution:

$$lseg_M(n,n) = \{emp_M\} \text{ for all } n \geq 0$$
$$lseg_M(0,m) = \emptyset \text{ for all } m \neq 0$$
$$lseg_M(n,m) = \{\ n \mapsto_M n_1 *_M n_1 \mapsto_M n_2 *_M \cdots *_M n_{k-1} \mapsto_M m$$
$$\qquad \mid k > 0, \text{ and } n_0 = n, n_1, n_2, \ldots, n_{k-1} > 0 \text{ all different and different from } m\}$$

Indeed, the first two equalities that need to be satisfied by any solution vacuously hold, while for the third all we need to note is that the "junk" maps where n is 0 or in the domain of a map in $lseg_M(n_1,m)$ are simply discarded by the map merge interpretation of $\_ * \_$.

Second, we claim that the above is the unique solution. Let $lseg_M : M_{Nat} \times M_{Nat} \to \mathcal{P}(M_{Map})$ be some solution satisfying the three equality constraints. It suffices to prove, by

induction on the size k of the domain of $h \in M_{Map}$ that: $h \in lseg_M(n,m)$ for $n,m \in M_{Nat}$ iff either $n = m$ and $h = emp_M$ (i.e., $k = 0$), or otherwise $n \neq 0$ and $n \neq m$ and $k > 0$ and there are distinct $n_0 = n$, n_1, ..., n_{k-1} distinct from m such that $h = (n \mapsto_M n_1 *_M n_1 \mapsto_M n_2 *_M \cdots *_M n_{k-1} \mapsto_M m)$. Since the maps in $lseg_M(n,m)$ when $n \neq 0$ and $n \neq m$ contain at least one binding, we conclude $k = 0$ can only happen iff $h \in lseg_M(n,n)$, and then $h = emp_M$. Now suppose $k > 0$, which can only happen iff $h \in lseg_M(n,m)$ for $n \neq 0$ and $n \neq m$, which can only happen iff $n \neq 0$ and $n \neq m$ and $h = n \mapsto_M n_1 *_M h_1$ for some $n_1 \geq 0$ and $h_1 \in lseg_M(n_1,m)$. It all follows now by the induction hypothesis applied to h_1.

It should be clear that patterns can be specified in many different ways. E.g., the first list pattern can also be specified with a single pattern:

$$list(x) = (x = 0 \wedge emp \vee \exists z \,.\, x \mapsto z * list(z))$$

We can similarly define more complex patterns, such as lists with data. But first, we specify a convenient operation for defining maps over contiguous keys, making use of a sequence data-type. The latter can be defined like in Section 6.1; for notational convenience, we take the freedom to use comma "," instead of "$\cdot$" for sequence concatenation in some places:

$$\_ \mapsto [\_] : Nat \times Seq \to Map \qquad \begin{array}{l} x \mapsto [\epsilon] = emp \\ x \mapsto [a, S] = x \mapsto a * (x+1) \mapsto [S] \end{array}$$

In our model M, we can take M_{Seq} to be the finite sequences of natural numbers, with ϵ_M and $\_ \cdot_M \_$ interpreted as the empty sequence and, respectively, sequence concatenation.

We can now define lists with data as follows:

$$\begin{array}{ll} list \in \Sigma_{Nat \times Seq, Map} & lseg \in \Sigma_{Nat \times Seq \times Nat, Map} \\ list(x, \epsilon) = (emp \wedge x = 0) & lseg(x, \epsilon, y) = (emp \wedge x = y) \\ list(x, n \cdot S) = \exists z \,.\, x \mapsto [n, z] * list(z, S) & lseg(x, n \cdot S, y) = \exists z \,.\, x \mapsto [n, z] * lseg(z, S, y) \end{array}$$

Note that, unlike in the case of lists without data, this time we have not required the side conditions $x \neq 0$ and $x \neq y$, respectively. The side conditions were needed in the former case because without them we can infer, e.g., $list(0) = \bot$ (from the second equation of *list*), which using the first equation would imply $emp = \bot$. However, they are not needed in the latter case because it is safe (and even desired) to infer $list(0, n \cdot S) = \bot$ for any n and S. We can show, using a similar approach like for lists without data, that the pattern $lseg(x, S, y)$ matches in M precisely the lists starting with x, exiting to y, and holding data sequence S.

We can similarly define other data-type specifications, such as trees with data:

$$\begin{array}{l} none : \to Tree \\ node : Nat \times Tree \times Tree \to Tree \\ tree \in \Sigma_{Nat \times Tree, Map} \\ \\ tree(0, none) = emp \\ tree(x, node(n,t_1,t_2)) = \exists y\, z \,.\, x \mapsto [n, y, z] * tree(y, t_1) * tree(z, t_2)) \end{array}$$

Therefore, in the model M of partial maps described above, there is a unique way to interpret *list* and *lseg*, namely as the expected linked lists and, respectively, linked list fragments. Fixing the interpretations of the basic mathematical domains, such as those of natural numbers, sequences, maps, etc., suffices in order to define interesting map patterns that appear in verification of heap properties of programs, in the sense that the axioms themselves uniquely define the desired data-types. No logic extensions (such as adding free models with induction/recursion principles as a matching logic equivalent to "recursive predicates", or least fixed-point constraints, or even fixed points of any kind) were needed

to define them. The defining axioms were precise enough to capture the intended concept in the intended model. Choosing the right basic mathematical domains is, however, crucial. For example, if we allow the maps in M_{Map} to have infinite domains then the list patterns without data above (the first ones) also include infinite lists. The lists with data cannot include infinite lists, because we only allow finite sequences. This would, of course, change if we allow infinite sequences, or streams, in the model. In that case, *list* and *lseg* would not admit unique interpretations anymore, because we can interpret them to be either all the finite domain lists, or both the finite and the infinite-domain lists. Writing patterns which admit the desired solution in the desired model suffices in practice; our reasoning techniques developed in the sequel allow us to derive properties that hold in all models satisfying the axioms, so any derived property is sound also for the intended model and interpretations.

9.3. **Separation Logic as an Instance of Matching Logic.** Consider the FOL fragment in Section 7, where the signature Σ includes the signature of maps in Section 9.2. Any additional FOL constructs, background theories, and/or built-in domains that one wants to consider in separation logic specifications, are handled as already explained in Sections 7 and 5.8. It is clear then that all the syntactic constructs of separation logic, except for the magic wand, $-\!\!*$, are given by the above matching logic signature. The magic wand, on the other hand, can be defined as the following derived construct:

$$\varphi_1 -\!\!* \varphi_2 \quad \equiv \quad \exists H : \mathit{Map} \,.\, H \wedge \lfloor H * \varphi_1 \rightarrow \varphi_2 \rfloor$$

Recall from Section 5.1 that $\lfloor \varphi \rfloor$ is $\top$ (it matches the entire set) iff its enclosed pattern φ is $\top$; otherwise, if φ does not match some elements, then $\lfloor \varphi \rfloor$ is $\bot$ (it matches nothing). In words, $\varphi_1 -\!\!* \varphi_2$ matches all maps h which merged with maps matching φ_1 yield only maps matching φ_2. Thanks to the notational convention that Booleans b, respectively usual predicates p, stand for equalities $b = \mathit{true}$, respectively $p = \top_{\mathit{Pred}}$ (Notation 5.27),

Any separation logic formula is a matching logic pattern of sort Map.

Semantically, note that separation logic hard-wires a particular model of maps. That is, its satisfaction relation $\models_{SL} \varphi$ is defined using a pre-defined universe of maps, which is conceptually the same as our model of maps in Section 9.2. Since separation logic extends FOL, its models are still allowed to instantiate the FOL part of its syntax in the usual FOL way, but the maps are rigid and the models cannot touch them. It is therefore not surprising that we also have to fix the maps in our matching logic models corresponding to the syntax described so far in order to faithfully capture separation logic semantically. For the rest of this section, we only consider models M for the matching logic specification above whose reduct to the syntax of maps is precisely the map domain in Section 9.2. We let $\mathit{Map} \models \varphi$ denote the resulting satisfaction relation: $\mathit{Map} \models \varphi$ iff $M \models \varphi$ for any model M like above.

In separation logic formulae, variables range only over the domains of data (e.g., natural numbers), but not over heaps/maps; for example, "$\exists H : \mathit{Map} \,.\, 1 \mapsto 2 * H$" is not a proper separation logic formula (although it is one in matching logic). Also, stores s are mappings of variables to particular values. In matching logic, variables range over all syntactic categories, including over heaps in our case, and valuations ρ can map such variables to any values in the model; for example, the variable H of sort *Map* in the pattern defining $-\!\!*$ above is a heap variable. Since separation logic formulae φ contain no heap variables, we can regard separation logic stores s as M-valuations with the property that $\overline{s}(\varphi)$ contains precisely the heaps which together with s satisfy the original separation logic formula φ. We prove this

in the next proposition showing that separation logic is not only syntactically an instance of matching logic (modulo notations in Section 5), but also semantically:

Proposition 9.2. *If φ is a separation logic formula, then $\models_{SL} \varphi$ iff $Map \models \varphi$.*

Proof. Like in the proofs of Propositions 4.1, 6.1, and 7.1, there is a bijection between the models of separation logic and the matching logic *Map*-models. The bijection works as described in the aforementioned propositions for the FOL part, and adds the map model in Section 9.2 to the resulting matching logic models. To keep the notation simple, we use the same name, M, to refer both to a separation logic model and to its corresponding matching logic model, remembering from the proofs of Propositions 4.1 and 6.1 that the latter's carrier of sort *Pred* is a singleton $\{\star\}$. The context makes it clear which one we are referring to.

We show by structural induction on the separation logic formula φ the more general result that for any store s and any heap h, we have $(s,h) \models_{SL} \varphi$ iff $h \in \overline{s}(\varphi)$.

If φ is a FOL formula then its desugared matching logic correspondent is $\varphi =^{Map}_{Pred} \top_{Pred}$ (Notation 7.2). Then $(s,h) \models_{SL} \varphi$ iff $s \models_{FOL} \varphi$ (Definition 9.1), iff $\overline{s}(\varphi) = \{\star\}$ (see proof of Proposition 7.1), iff $\overline{s}(\varphi) = \overline{s}(\top_{Pred})$, iff $\overline{s}(\varphi =^{Map}_{Pred} \top_{Pred}) = M_{Map}$ (by Proposition 5.9), iff $h \in \overline{s}(\varphi =^{Map}_{Pred} \top_{Pred})$ (Proposition 5.9 again: equality is interpreted as either M_{Map} or $\emptyset$).

Conjunction and negation are trivial. Existential quantification: $(s,h) \models_{SL} \exists x \,.\, \varphi$ iff there exists some $a \in M$ of appropriate (non-heap) sort such that $(s[a/x], h) \models \varphi$, iff $h \in \overline{s[a/x]}(\varphi)$ (induction hypothesis), iff $h \in \bigcup\{\overline{s'}(\varphi) \mid s' : Var \to M,\ s'{\restriction}_{Var\setminus\{x\}} = s{\restriction}_{Var\setminus\{x\}}\}$, iff $h \in \overline{s}(\exists x \,.\, \varphi)$. We next discuss the heap-related constructs of separation logic.

If $\varphi \equiv emp$ then $(s,h) \models_{SL} emp$ iff $h = emp_M$, iff $h \in \{emp_M\}$, iff $h \in \overline{s}(emp)$.

If $\varphi \equiv e_1 \mapsto e_2$ then $(s,h) \models_{SL} e_1 \mapsto e_2$ iff $Dom(h) = \overline{s}(e_1) \neq 0$ and $h(\overline{s}(e_1)) = \overline{s}(e_2)$ (Definition 9.1), iff h is the partial-domain map $\overline{s}(e_1) \mapsto_M \overline{s}(e_2)$ (which is undefined when $\overline{s}(e_1) = 0$—see Section 9.2), iff $h \in \overline{s}(e_1 \mapsto e_2)$.

If $\varphi \equiv \varphi_1 * \varphi_2$ then $(s,h) \models_{SL} \varphi_1 * \varphi_2$ iff there exist h_1 and h_2 of disjoint domains such that $h = h_1 *_M h_2$ (the merge of h_1 and h_2, which is a partial function on maps—see Definition 9.1 and Section 9.2) and $(s,h_1) \models_{SL} \varphi_1$ and $(s,h_2) \models_{SL} \varphi_2$, iff $h = h_1 *_M h_2$ and $h_1 \in \overline{s}(\varphi_1)$ and $h_2 \in \overline{s}(\varphi_2)$ (induction hypothesis), iff $h \in \overline{s}(\varphi_1) *_M \overline{s}(\varphi_2)$, iff $h \in \overline{s}(\varphi_1 * \varphi_2)$.

The only case left is the "magic wand", $\varphi \equiv \varphi_1 \mathrel{-\!\!*} \varphi_2$:

$$
\begin{array}{ll}
 & h \in \overline{s}(\varphi_1 \mathrel{-\!\!*} \varphi_2) \\
\text{iff} & h \in \overline{s}(\exists H \,.\, H \wedge \lfloor H * \varphi_1 \to \varphi_2 \rfloor) \\
\text{iff} & \{h\} *_M \overline{s}(\varphi_1) \subseteq \overline{s}(\varphi_2) \\
\text{iff} & h * h_1 \in \overline{s}(\varphi_2) \text{ for any } h_1 \in \overline{s}(\varphi_1) \text{ compatible with } h \\
\text{iff} & (s, h * h_1) \models_{SL} \varphi_2 \text{ for any } h_1 \text{ compatible with } h \text{ such that } (s,h_1) \models_{SL} \varphi_1 \\
 & \text{(previous step followed by the induction hypothesis)} \\
\text{iff} & (s,h) \models_{SL} \varphi_1 \mathrel{-\!\!*} \varphi_2
\end{array}
$$

The proof is complete. □

Although matching logic is complete (Section 11), so its validity $\models$ is semi-decidable, while results in [21, 1] state that the validity problem in separation logics is not semi-decidable, note that there is no conflict here because we restricted matching logic validity to *Map*-models. As an analogy, it is well-known that the validity of predicate logic formulae can be arbitrarily hard when particular (and not all) models are concerned. All the above says is that the results in [21, 1] carry over to the particular matching logic theory restricted to *Map*-models discussed in this section. Most likely one can obtain even more general instances of the results [21, 1] for matching logic, but that is beyond the scope of this paper.

The loose-model approach of matching logic is in sharp technical, but not conceptual, contrast to separation logic. In separation logic, the syntax of maps and separation constructs is part of the syntax of the logic itself, and the model of maps is intrinsically integrated within the semantics of the logic: its satisfaction relation is defined in terms of a fixed syntax and the fixed model of the basic domains (maps, sequences, etc.). Then specialized proof rules and theorem provers need to be devised. If any changes to the syntax or semantics are desired, for example adding a new stack, or an I/O buffer, etc., then a new logic is obtained. Proof rules and theorem provers may need to change as the logic changes. In matching logic, the basic ingredients of separation logic form one particular specification, with its particular signature and pattern axioms, together with particular but carefully chosen models. This enables us to use generic first-order reasoning in matching logic (Section 11), as well as theorem provers or SMT solvers for reasoning about the intended models. Nevertheless, several high performance automated provers for separation logics have been developed, e.g. [8, 2, 15, 64, 9, 65, 72, 75, 73], while there are no automated provers available for matching logic yet. A technical challenge, left for future work, is to investigate the techniques and algorithms underlying the existing separation logic provers and to generalize them if possible to work with matching logic in general or at least with common fragments of it.

Like for modal logic (Section 8), the result above in combination with the reduction of matching logic to predicate logic with equality in Section 10 yields a translation from separation logic to predicate logic with equality. Note that many of the separation logic provers above are implicitly or explicitly based on translations to FOL, and specific translations to FOL or fragments of it have been already studied [20, 22, 12]. Like for modal logic (Section 8), our goal in this section was not to propose yet another translation. Our goal was to show how separation logic can be framed as a matching logic specification both syntactically and semantically, without any translation (but only with syntactic sugar notations). Such results can help us better understand both logics, as well as their strengths and limitations.

10. Matching Logic: Reduction to Predicate Logic with Equality

It is known that FOL formulae can be translated into equivalent predicate logic with equality formulae (i.e., no function or constant symbols—see Section 4), by replacing all functions with their graph relations (see, e.g., [56]). Specifically, function symbols $\sigma : s_1 \times \cdots \times s_n \to s$ are replaced with predicate symbols $\pi_\sigma : s_1 \times \cdots \times s_n \times s$, and then terms are transformed into formulae by adding existential quantifiers for subterms. Let us define such a translation, say PL. It takes each FOL predicate $\pi(t_1, \ldots, t_n)$ into a pure predicate logic formula as follows:

$$PL(\pi(t_1, \ldots, t_n)) = \exists r_1 \cdots r_n \,.\, PL_2(t_1, r_1) \wedge \cdots \wedge PL_2(t_n, r_n) \wedge \pi(r_1, \ldots, r_n)$$

where $PL_2(t, r)$ is a translation of term t into a predicate stating that t equals variable r:

$$\begin{array}{rcl} PL_2(x, r) & = & (x = r) \\ PL_2(\sigma(t_1, \ldots, t_n), r) & = & \exists r_1 \cdots \exists r_n \,.\, PL_2(t_1, r_1) \wedge \cdots \wedge PL_2(t_n, r_n) \wedge \pi_\sigma(r_1, \ldots, r_n, r) \end{array}$$

Axioms stating that the predicate symbols $\pi_\sigma : s_1 \times \cdots \times s_n \times s$ associated to function symbols $\sigma : s_1 \times \cdots \times s_n \to s$ are interpreted as total function relations are also needed:

$$\forall x_1 : s_1, \ldots, x_n : s_n \,.\, \exists y : s \,.\, \forall z : s \,.\, (\pi_\sigma(x_1, \ldots, x_n, z) \leftrightarrow y = z)$$

We can similarly translate matching logic patterns into equivalent predicate logic formulae. Consider predicate logic with equality (and no function or constant symbols) whose satisfaction relation is $\models^{=}_{PL}$. For matching logic signature (S, Σ), let (S, Π_Σ) be the predicate

logic signature with $\Pi_\Sigma = \{\pi_\sigma : s_1 \times \cdots \times s_n \times s \mid \sigma \in \Sigma_{s_1 \ldots s_n, s}\}$, like above but without the axioms stating that these predicates have a functional interpretation in models (because the matching logic symbols need not be interpreted as functions). We define the translation PL of matching logic (S, Σ)-patterns into predicate logic (S, Π_Σ)-formulae inductively:

$$
\begin{aligned}
PL(\varphi) &= \forall r \,.\, PL_2(\varphi, r) \\
\\
PL_2(x, r) &= (x = r) \\
PL_2(\sigma(\varphi_1, \ldots, \varphi_n), r) &= \exists r_1 \cdots \exists r_n \,.\, PL_2(\varphi_1, r_1) \wedge \cdots \wedge PL_2(\varphi_n, r_n) \wedge \pi_\sigma(r_1, \ldots, r_n, r) \\
PL_2(\neg\varphi, r) &= \neg PL_2(\varphi, r) \\
PL_2(\varphi_1 \wedge \varphi_2, r) &= PL_2(\varphi_1, r) \wedge PL_2(\varphi_2, r) \\
PL_2(\exists x \,.\, \varphi, r) &= \exists x \,.\, PL_2(\varphi, r) \\
\\
PL(\{\varphi_1, \ldots, \varphi_n\}) &= \{PL(\varphi_1), \ldots, PL(\varphi_n)\}
\end{aligned}
$$

The predicate logic formula $PL_2(\varphi, r)$ captures the intuition that "r matches φ". The top transformation above, $PL(\varphi) = \forall r \,.\, PL_2(\varphi, r)$, is different from (and simpler than) the corresponding translation of predicates from FOL to predicate logic. It captures the intuition that the pattern φ is valid iff it is matched by *all* values r. Then the following result holds:

Proposition 10.1. *If F is a set of patterns and φ is a pattern, $F \models \varphi$ iff $PL(F) \models^{=}_{PL} PL(\varphi)$.*

Proof. It suffices to show that there is a bijective correspondence between matching logic (S, Σ)-models M and predicate logic (S, Π_Σ)-models M', such that $M \models \varphi$ iff $M' \models^{=}_{PL} PL(\varphi)$ for any (S, Σ)-pattern φ. The bijection is defined as follows:

- $M'_s = M_s$ for each sort $s \in S$;
- $\pi_{\sigma M'} \subseteq M_{s_1} \times \cdots \times M_{s_n} \times M_s$ with $(a_1, \ldots, a_n, a) \in \pi_{\sigma M'}$ iff $\sigma_M : M_{s_1} \times \cdots \times M_{s_n} \to \mathcal{P}(M_s)$ with $a \in \sigma_M(a_1, \ldots, a_n)$.

To show $M \models \varphi$ iff $M' \models^{=}_{PL} PL(\varphi)$, it suffices to show $a \in \overline{\rho}(\varphi)$ iff $\rho[a/r] \models^{=}_{PL} PL_2(\varphi, r)$ for any $\rho : Var \to M$, which follows easily by structural induction on φ. □

It is informative to translate the definedness and equality patterns in Sections 5.1 and Section 5.2 using the above, and especially to sanity check that the equality pattern of matching logic indeed translates to the equality predicate of predicate logic with equality. Recall that the definedness symbols were axiomatized with pattern axioms $\lceil x \rceil$, and that we assumed them always available (Assumption 5.1). Then $PL(\lceil x \rceil)$ is $\forall r \,.\, \pi_{\lceil \_ \rceil}(x, r)$. We can drop the universal quantifier and therefore assume $\pi_{\lceil \_ \rceil}(x, r)$ as an axiom formula in the translated predicate logic specification. Let us now show that the matching logic equality $x = y$, which is syntactic sugar for $\neg\lceil\neg(x \leftrightarrow y)\rceil$, translates to the equality $x = y$ in predicate logic. Applying the translation above, we get $PL(x = y)$ is $\forall r \,.\, \neg(\exists r_1 \,.\, \neg(x = r_1 \leftrightarrow y = r_1) \wedge \pi_{\lceil \_ \rceil}(r_1, r))$, which is equivalent, in predicate logic with equality, to $\forall r \,.\, \forall r_1 \,.\, (x = r_1 \leftrightarrow y = r_1)$, which is further equivalent to $x = y$. Similarly, we can show that the translation of the equational pattern stating that σ is functional, namely $\forall x_1 \ldots x_n \,.\, \exists y \,.\, \sigma(x_1, \ldots, x_n) = y$, indeed corresponds to the predicate logic formula $\forall x_1, \ldots, x_n \,.\, \exists y \,.\, \forall z \,.\, (\pi_\sigma(x_1, \ldots, x_n, z) \leftrightarrow y = z)$, as expected. We leave this as an exercise to the interested reader.

Proposition 10.1 gives us a sound and complete procedure for matching logic reasoning: translate the specification (S, Σ, F) and pattern to prove φ into the predicate logic specification $(S, \Pi_\Sigma, PL(F))$ and formula $PL(\varphi)$, respectively, and then derive it using the sound and complete proof system of predicate logic. However, translating patterns to predicate logic formulae makes reasoning harder not only for humans, but also for computers, since

new quantifiers are introduced. For example,

$$(1 \mapsto 5 * 2 \mapsto 0 * 7 \mapsto 9 * 8 \mapsto 1) \rightarrow \mathit{list}(7, 9 \cdot 5)$$

discussed and proved in a few steps in Section 11, translates into the following formula (to keep it small, we do not translate the numbers), which takes dozens, if not hundreds of steps to prove using the predicate logic proof system:

$$\begin{array}{l}\forall r \,.\, (\exists r_1 \,.\, \exists r_2 \,.\, \pi_{\mapsto}(1, 5, r_1) \wedge (\exists r_3 \,.\, \exists r_4 \,.\, \pi_{\mapsto}(2, 0, r_3) \wedge (\exists r_5 \,.\, \exists r_6 \,.\, \pi_{\mapsto}(7, 9, r_5) \wedge \pi_{\mapsto}(8, 1, r_6) \\ \qquad \wedge \pi_*(r_5, r_6, r_4)) \wedge \pi_*(r_3, r_4, r_2)) \wedge \pi_*(r_1, r_2, r)) \rightarrow \exists r_7 \,.\, \pi.(9, 5, r_7) \wedge \pi_{\mathit{list}}(7, r_7, r))\end{array}$$

What we would like is to reason directly with matching logic patterns, the same way we reason directly with terms in FOL without translating them to predicate logic.

11. Matching Logic: Sound and Complete Deduction

In Figure 5, we propose a sound and complete proof system for matching logic (under Assumption 5.1). The first group of rules/axioms are those of FOL with equality, discussed and proved sound in Section 2 (predicate logic: Proposition 2.8), Section 5.2 (equational: Proposition 5.9), and Section 5.4 (FOL Substitution, called Term Substitution there: Corollary 5.23), with a slightly generalized Substitution axiom that we call Functional Substitution (discussed below), which requires another axiom (shown sound by Corollary 5.19), called Functional Variable, stating that variables are functional. The second group of rules/axioms are about membership and were proved sound in Section 5.3 (Proposition 5.14).

Substitution must be used with care. Recall FOL's Substitution: $(\forall x \,.\, \varphi) \rightarrow \varphi[t/x]$. Since matching logic makes no syntactic distinction between terms and predicates, we would like to have a proof system that does not make such a distinction either. Ideally, since terms and predicates are particular patterns, we would like to take the proof system of FOL with equality and turn it into a proof system for matching logic by simply replacing "predicate" and "term" with "pattern". This actually worked for all the rules and axioms, except for Substitution: $(\forall x \,.\, \varphi) \rightarrow \varphi[t/x]$. Unfortunately, Substitution is not sound if we replace t with any pattern. For example, let φ be $\exists y \,.\, x = y$ (Corollary 5.19). If FOL's Substitution were sound for arbitrary patterns φ' instead of t, then the formula $\exists y \,.\, \varphi' = y$, stating that φ' is a functional pattern (i.e., it evaluates to a unique element for any valuation: Definition 5.16), would be valid for any pattern φ'. That is, any pattern would be functional, which is neither true nor desired (e.g., $\top$ evaluates to the total set, $\bot$ to the empty set, etc.).

Nevertheless, as proved in Corollary 5.23, Substitution stays sound if t is a term pattern (Definition 5.21), that is, a pattern build with only functional symbols (interpreted as functions in all models) and no other constructs: $\models (\forall x \,.\, \varphi) \rightarrow \varphi[t/x]$ holds if φ is any pattern but t is a term pattern. It turns out that the fact that t is built with only functional symbols is irrelevant, and all that matters is that t is a functional pattern (all term patterns are functional: Corollary 5.22). We therefore generalize the Term Substitution axiom:

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

This is more general than the original Substitution in FOL (which allowed only predicates for φ) and than Term Substitution (Corollary 5.23): it can also apply when φ' is not a term pattern but can be proved to be functional. It is interesting to note that a similar modification of Substitution was needed in the context of *partial* FOL (PFOL) [35], where the interpretations of functional symbols are partial functions, so terms may be undefined; axiom PFOL5 in [35] requires φ' to be *defined* in the Substitution rule, and several rules

FOL axioms and rules:

1. $\vdash$ propositional tautologies
2. Modus Ponens: $\vdash \varphi_1$ and $\vdash \varphi_1 \to \varphi_2$ imply $\vdash \varphi_2$
3. $\vdash (\forall x \,.\, \varphi_1 \to \varphi_2) \to (\varphi_1 \to \forall x \,.\, \varphi_2)$ when $x \notin FV(\varphi_1)$
4. Universal Generalization: $\vdash \varphi$ implies $\vdash \forall x \,.\, \varphi$
5. Functional Substitution: $\vdash (\forall x \,.\, \varphi) \wedge (\exists y \,.\, \varphi' = y) \to \varphi[\varphi'/x]$

5'. Functional Variable: $\vdash \exists y.\, x = y$

6. Equality Introduction: $\vdash \varphi = \varphi$
7. Equality Elimination: $\vdash \varphi_1 = \varphi_2 \wedge \varphi[\varphi_1/x] \to \varphi[\varphi_2/x]$

Membership axioms and rules:

8. $\vdash \forall x \,.\, x \in \varphi$ iff $\vdash \varphi$
9. $\vdash x \in y = (x = y)$ when $x, y \in \mathit{Var}$
10. $\vdash x \in \neg\varphi = \neg(x \in \varphi)$
11. $\vdash x \in \varphi_1 \wedge \varphi_2 = (x \in \varphi_1) \wedge (x \in \varphi_2)$
12. $\vdash (x \in \exists y.\varphi) = \exists y.(x \in \varphi)$, with x and y distinct
13. $\vdash x \in \sigma(\varphi_1,..,\varphi_{i-1},\varphi_i,\varphi_{i+1},..,\varphi_n) = \exists y.(y \in \varphi_i \wedge x \in \sigma(\varphi_1,..,\varphi_{i-1},y,\varphi_{i+1},..,\varphi_n))$

Figure 5: Sound and complete proof system of matching logic.

for proving definedness are provided. Note that our condition $\exists y \,.\, \varphi' = y$ is equivalent to definedness in the special case of PFOL, and that, thanks to the definability of equality in matching logic, we do not need any special axiomatic or rule support for proving definedness.

We have made no effort to minimize the number of rules and axioms in our proof system in Figure 5. On the contrary, our approach was to include all the rules and axioms that turned out to be useful in proof derivations, especially if they already existed in FOL. Moreover, we preferred to frame "unexpected" properties of matching logic as axioms or proof rules, so that users of the proof system are fully aware of them. For example, we could have merged the Functional Substitution and Functional Variable axioms into the conventional predicate logic Substitution ((5) in Proposition 2.8) or the FOL Term Substitution (Corollary 5.23), but we preferred not to, because we want the user of our proof system to be fully aware of the fact that they cannot substitute arbitrary patterns for variables; they should first prove that the pattern is functional. Additionally, our Functional Substitution is more general, in that it applies in more instances, so proof derivations are shorter.

Proposition 11.1. *With the proof system in Figure 5, the following are derivable:*

(1) *Predicate Logic Substitution ((5) in Proposition 2.8):* $\vdash (\forall x \,.\, \varphi) \to \varphi[y/x]$
(2) *Term patterns are functional (Corollary 5.22):* $\vdash \exists y \,.\, t = y$ *for any term pattern* t
(3) *Term Substitution (Corollary 5.23):* $\vdash (\forall x \,.\, \varphi) \to \varphi[t/x]$

Proof. By propositional calculus reasoning, which is subsumed by our proof system (1. and 2. in Figure 5), for any patterns A, B, and C, if $\vdash A \wedge B \to C$ and $\vdash B$ then $\vdash A \to C$. To prove (1), pick A as $\forall x \,.\, \varphi$, B as $\exists z \,.\, y = z$, C as $\varphi[y/x]$. Then $\vdash A \wedge B \to C$ by Functional Substitution and $\vdash B$ by Functional Variable, so $\vdash A \to C$, i.e., $\vdash (\forall x \,.\, \varphi) \to \varphi[y/x]$.

We prove (2) and (3) together, by structural induction on t. If t is a variable then they follow by Functional Variable and, respectively, by (1). Suppose that t is $\sigma(t_1, \ldots, t_n)$ for some functional symbol σ, i.e., one for which we have an axiom $\exists y \,.\, \sigma(x_1, \ldots, x_n) = y$ (Definition 5.21), and for some appropriate term patterns t_1, ..., t_n. By the induction

hypothesis of (2), we have $\vdash \exists y_1 \,.\, t_1 = y_1, \ldots, \vdash \exists y_n \,.\, t_n = y_n$. By the induction hypothesis on (3) with x as x_1 and φ as $\exists y \,.\, \sigma(x_1, \ldots, x_n) = y$, we derive

$$\vdash (\forall x_1 \,.\, \exists y \,.\, \sigma(x_1, \ldots, x_n) = y) \rightarrow \exists y \,.\, \sigma(t_1, \ldots, x_n) = y$$

Since $\vdash \forall x_1 \,.\, \exists y \,.\, \sigma(x_1, \ldots, x_n) = y$ by the functionality axiom of σ and Universal Generalization, we derive $\vdash \exists y \,.\, \sigma(t_1, x_2, \ldots, x_n) = y$. By the induction hypothesis on (3) with x as x_2 and φ as $\exists y \,.\, \sigma(t_1, x_2, \ldots, x_n) = y$, we derive $\vdash (\forall x_2 \,.\, \exists y \,.\, \sigma(t_1, x_2, \ldots, x_n) = y) \rightarrow \exists y \,.\, \sigma(t_1, t_2, \ldots, x_n) = y$. Since $\vdash \forall x_2 \,.\, \exists y \,.\, \sigma(t_1, x_2, \ldots, x_n) = y$ by the previous derivation and Universal Generalization, we derive $\vdash \exists y \,.\, \sigma(t_1, t_2, x_3, \ldots, x_n) = y$. Iterating this process for all the arguments of σ, we eventually derive $\vdash \exists y \,.\, \sigma(t_1, \ldots, t_n) = y$, that is, $\vdash \exists y \,.\, t = y$. The only thing left is to prove (3). We prove it similarly to (1), using (2): in the propositional calculus property at the beginning of the proof, pick A as $\forall x \,.\, \varphi$, B as $\exists y \,.\, t = y$, and C as $\varphi[t/x]$. Then $\vdash A \wedge B \rightarrow C$ by Functional Substitution and $\vdash B$ by (2) above, so $\vdash A \rightarrow C$, i.e., $\vdash (\forall x \,.\, \varphi) \rightarrow \varphi[t/x]$. □

Our approach to obtain a sound and complete proof system for matching logic is to build upon its reduction to predicate logic with equality in Section 10. Specifically, to use Proposition 10.1 and the complete proof system of predicate logic with equality. Given a matching logic signature (S, Σ), let (S, Π_Σ) be the predicate logic (with equality) signature obtained like in Section 10. Besides the PL translation there, we also define a backwards translation ML of predicate logic with equality (S, Π_Σ)-formulae into (S, Σ)-patterns:

$$\begin{array}{rcl} ML(x = r) & = & x = r \\ ML(\pi_\sigma(r_1, \ldots, r_n, r)) & = & r \in \sigma(r_1, \ldots, r_n) \\ ML(\neg\psi) & = & \neg ML(\psi) \\ ML(\psi_1 \wedge \psi_2) & = & ML(\psi_1) \wedge ML(\psi_2) \\ ML(\exists x \,.\, \psi) & = & \exists x \,.\, ML(\psi) \\ ML(\{\psi_1, \ldots, \psi_n\}) & = & \{ML(\psi_1), \ldots, ML(\psi_n)\} \end{array}$$

Recall from Section 5.2 that we assume equality and membership in all specifications.

Theorem 11.2. *The proof system in Figure 5 is sound and complete: $F \models \varphi$ iff $F \vdash \varphi$.*

Proof. Propositions 2.8 and 5.10 showed the soundness of all rules except for Substitution. Corollary 5.23 showed the soundness of the stronger Term Substitution. To show the soundness of Functional Substitution, we show $\overline{\rho}((\forall x \,.\, \varphi) \wedge (\exists y \,.\, \varphi' = y)) \subseteq \overline{\rho}(\varphi[\varphi'/x])$ for any model M and valuation $\rho : Var \rightarrow M$. Let s be the sort of φ and s' be the sort of φ'. We have $\overline{\rho}((\forall x \,.\, \varphi) \wedge (\exists y \,.\, \varphi' = y)) = \bigcap\{\overline{\rho'}(\varphi) \mid \rho' \restriction_{Var\setminus\{x\}} = \rho \restriction_{Var\setminus\{x\}}\} \cap \bigcup\{M_s \mid \rho' \restriction_{Var\setminus\{x\}} = \rho \restriction_{Var\setminus\{x\}},\ \overline{\rho'}(\varphi') = \{\rho'(y)\}\}$. Since $y \notin FV(\varphi')$, it follows that $\overline{\rho'}(\varphi') = \overline{\rho}(\varphi')$. Therefore, all we have to show is the following: if $\overline{\rho}(\varphi') = \{a\}$ for some $a \in M_{s'}$ then $\bigcap\{\overline{\rho'}(\varphi) \mid \rho' \restriction_{Var\setminus\{x\}} = \rho \restriction_{Var\setminus\{x\}}\} \subseteq \overline{\rho}(\varphi[\varphi'/x])$. This holds because $\overline{\rho}(\varphi[\varphi'/x]) = \overline{\rho[a/x]}(\varphi)$.

We now show the completeness. First, note that Proposition 10.1 and the completeness of predicate logic imply that $F \models \varphi$ iff $PL(F) \vdash^{=}_{PL} PL(\varphi)$. Second, note that $PL(F) \vdash^{=}_{PL} PL(\varphi)$ implies $ML(PL(F)) \vdash ML(PL(\varphi))$, because the ML translation only replaces predicates $\pi_\sigma(r_1, \ldots, r_n, r)$ with $r \in \sigma(r_1, \ldots, r_n)$ and the proof rules of predicate logic, except for Substitution, are a subset of the proof rules in Figure 5, while the predicate logic Substitution is derivable in matching logic ((1) in Proposition 11.1). Third, notice that the completeness result holds if we can show $F \vdash \varphi$ iff $F \vdash ML(PL(\varphi))$ for any pattern φ: indeed, if that is the case then $F \vdash ML(PL(F))$, which together with $ML(PL(F)) \vdash ML(PL(\varphi))$ implies $F \vdash ML(PL(\varphi))$, which further implies $F \vdash \varphi$.

Let us now prove that $F \vdash \varphi$ iff $F \vdash ML(PL(\varphi))$ for any pattern φ. We first show $\vdash r \in \varphi = ML(PL_2(\varphi, r))$ by induction on φ. The cases $\varphi \equiv x$, $\varphi \equiv \neg\varphi'$, $\varphi \equiv \varphi_1 \wedge \varphi_2$, and $\varphi \equiv \exists y.\varphi'$ are immediate consequences of the axioms 9-12 in Figure 5, using the induction hypothesis and Equality Elimination (rule 7). For the case $\varphi \equiv \sigma(\varphi_1, \ldots, \varphi_n)$, we can first derive $\vdash ML(PL_2(\varphi, r)) = \exists r_1 \cdots \exists r_n \,.\, r_1 \in \varphi_1 \wedge \cdots \wedge r_n \in \varphi_n \wedge r \in \sigma(r_1, \ldots, r_n)$ using the induction hypothesis and Equality Elimination, and then $\vdash r \in \varphi = \exists r_1 \cdots \exists r_n \,.\, r_1 \in \varphi_1 \wedge \cdots \wedge r_n \in \varphi_n \wedge r \in \sigma(r_1, \ldots, r_n)$ using axiom 13 in Figure 5 and conventional FOL reasoning. Therefore, $\vdash r \in \varphi = ML(PL_2(\varphi, r))$. Our result now follows by proof rules 8 in Figure 5, since $ML(PL(\varphi)) \equiv \forall r \,.\, ML(PL_2(\varphi, r))$. □

As an example, let us informally use the proof system in Figure 5 together with the axiom patterns in Section 9.2, to derive $(1 \mapsto 5 * 2 \mapsto 0 * 7 \mapsto 9 * 8 \mapsto 1) \to \mathit{list}(7, 9 \cdot 5)$. For simplicity, like in separation logic, let us assume that the axioms of commutativity, associativity and idempotence of $\_ * \_$ are axiom *schemas*, so we do not need to explicitly use the substitution rule to instantiate them; in a mechanical verification setting, their soundness as schemas can be proved separately from the basic axioms.

Recall the following axiom patterns about linked lists with data from Section 9.2:

$$x \mapsto [\epsilon] = \mathit{emp} \qquad \mathit{list}(x, \epsilon) = (\mathit{emp} \wedge x = 0)$$
$$x \mapsto [a, S] = x \mapsto a * (x + 1) \mapsto [S] \qquad \mathit{list}(x, n \cdot S) = \exists z \,.\, x \mapsto [n, z] * \mathit{list}(z, S)$$

Using the left axioms, axioms for sequences in Section 6.1, and axioms of maps, by Functional Substitution and Equality Elimination (Figure 5) we derive $\vdash 1 \mapsto 5 * 2 \mapsto 0 = 1 \mapsto [5, 0]$ and $\vdash 7 \mapsto 9 * 8 \mapsto 1 = 7 \mapsto [9, 1]$, respectively. By the first axiom for *list* above, $\vdash \mathit{list}(0, \epsilon) = \mathit{emp}$. Note that Functional Substitution is equivalent to $\vdash \varphi[\varphi'/y] \wedge (\exists y \,.\, \varphi' = y) \to (\exists x \,.\, \varphi)$ (by propositional reasoning, e.g., $A \to B = \neg B \to \neg A$), so we get $\vdash 1 \mapsto [5, 0] * \mathit{list}(0, \epsilon) \to (\exists z \,.\, 1 \mapsto [5, z] * \mathit{list}(z, \epsilon))$, which by the second axiom of *list* above yields $\vdash 1 \mapsto [5, 0] * \mathit{list}(0, \epsilon) \to \mathit{list}(1, 5)$. Following similar reasoning for the other binding, we can construct the following (informal) proof derivation:

$$\begin{array}{lll}
 & 1 \mapsto 5 * 2 \mapsto 0 * 7 \mapsto 9 * 8 \mapsto 1 & \\
= & 1 \mapsto [5, 0] * 7 \mapsto [9, 1] & \\
= & 1 \mapsto [5, 0] * \mathit{list}(0, \epsilon) * 7 \mapsto [9, 1] & \text{(structural framing—Proposition 2.10)} \\
\to & (\exists z \,.\, 1 \mapsto [5, z] * \mathit{list}(z, \epsilon)) * 7 \mapsto [9, 1] & \\
= & \mathit{list}(1, 5 \cdot \epsilon) * 7 \mapsto [9, 1] & \\
= & \mathit{list}(1, 5) * 7 \mapsto [9, 1] & \\
\to & \exists z \,.\, 7 \mapsto [9, z] \wedge \mathit{list}(z, 5) & \\
= & \mathit{list}(7, 9 \cdot 5) &
\end{array}$$

When applying structural framing (Proposition 2.10) above, we assumed the completeness of the matching logic proof system (Theorem 11.2). It is an insightful exercise to directly prove Proposition 2.10 with $\vdash$ instead of $\models$, without using the completeness theorem but only the proof rules in Figure 5 (hint: use the membership rules).

The example proof above was neither difficult nor unexpected, and it followed almost the same steps as the corresponding separation logic proof. Indeed, in spite of matching logic's simplicity (recall that its syntax is even simpler than that of FOL: Definition 2.1) and domain-independence, it has the necessary expressiveness and capability to carry out proof derivations for particular domains given as matching logic specifications that are as abstract and intuitive as in logics specifically crafted for those domains. Additionally, its patterns are expressive enough to capture complex structural and logical properties about program

configurations, at the same time giving us the peace of mind that any such properties are derivable with a uniform, domain-independent proof system.

12. Additional Related Work

Matching logic builds upon intuitions from and relates to at least five important logical frameworks: (1) *Relation algebra (RA)* (see, e.g., [89]), noticing that our interpretations of symbols as functions to powersets are equivalent to relations; although our interpretation of symbols captures better the intended meaning of pattern and matching, and our proof system is quite different from that of RA, like with FOL we expect a tight relationship between matching logic and RA, which is left as future work; (2) *Partial FOL* (see, e.g., [35] for a recent work and a survey), noticing that our interpretations of symbols into powersets are more general than partial functions (Section 5.2 shows how we defined definedness); (3) *Separation logics (SL)* (see, e.g.,[68]), which we discussed in Section 9; and (4) Precursors of matching logic in [81, 84, 85, 86, 80, 27], which proposed the pattern idea by extending FOL with particular "configuration" terms (grayed box below is the only change to FOL):

$$\begin{array}{rcl} t_s & ::= & x \in \mathit{Var}_s \quad | \quad \sigma(t_1, \ldots, t_n) \text{ with } \sigma \in \Sigma_{s_1 \ldots s_n, s} \\ \varphi & ::= & \pi(x_1, \ldots, x_n) \text{ with } \pi \in \Pi_{s_1 \ldots s_n} \quad | \quad \neg\varphi \quad | \quad \varphi \wedge \varphi \quad | \quad \exists x.\varphi \\ & | & \boxed{t \in T_{\Sigma, \mathit{Cfg}}(X)} \end{array}$$

where $T_{\Sigma,\mathit{Cfg}}(X)$ is the set of terms of a special sort *Cfg* (from "configurations") over variables in set X. To avoid terminology conflicts, we here strengthen the proposal in [79] to call the variant above *topmost matching logic* from here on. Topmost matching logic can trivially be desugared into FOL with equality by regarding a particular pattern predicate $t \in T_{\Sigma,\mathit{Cfg}}(X)$ as syntactic sugar for "(current state/configuration is) equal to t", i.e., $\Box = t$. One major limitation of topmost matching logic, which motivated the generalization in [79] with full details added in this paper, is that its restriction to patterns of sort *Cfg* prevented us to define local patterns (e.g., the heap list pattern) and perform local reasoning. They had to be defined globally, as patterns of sort *Cfg* with structural frames for everything else except their target cell (e.g., the heap), which was not only more verbose but also less modular.

The basic idea of regarding terms with variables as sentences/patterns that are satisfied/matched by ground terms, goes back to [59] and it was further studied in [58, 88, 36, 74, 62]. Furthermore, terms enriched with Boolean conditions over their variables, called *constrained terms*, were studied in [23], together with their relation to narrowing. These approaches allow certain Boolean algebra operations to be applied to patterns, and study the expressiveness of such operations w.r.t. the languages of ground terms that they define, in particular conditions under which negation can be eliminated. In addition to Boolean algebra operations and conditions on terms with variables, matching logic also allows quantification over variables, as well as using the resulting patterns nested inside other patterns. The richer syntax of patterns in matching logic is motivated by needs to specify complex structures with mixed constraints over program configurations, as shown in Section 2.2. Also, matching logic allows models with any data, not only term models, interprets symbols as relations with the axiomatic capability to constrain them as functions, and organizes the patterns and their models in a logic that admits a sound and complete proof system.

The idea of regarding terms as patterns is also reminiscent of *pattern calculus* [53], although note that matching logic's patterns are intended to express and reason about static properties of data-structures or program configurations, while pattern calculi are aimed at

generally and compactly expressing computations and dynamic behaviors of systems. So far we used rewriting to define dynamic semantics; it would be interesting to explore the combination of pattern calculus and matching logic for language semantics and reasoning.

13. Conclusion and Future Work

Matching logic is a sound and complete FOL variant that makes no distinction between function and predicate symbols. Its formulae, called patterns, mix symbols, logical connectives and quantifiers, and evaluate in models to sets of values, those that "match" them, instead of just one value as terms do or a truth value as predicates do in FOL. Equality can be defined and several important variants of FOL fall as special fragments. Separation logic can be framed as a matching logic theory within the particular model of partial finite-domain maps, and heap patterns can be elegantly specified using equations. Matching logic allows spatial specification and reasoning anywhere in a program configuration, and for any language, not only in the heap or other particular and fixed semantic components.

We made no efforts to minimize the number of rules in our proof system (Figure 5), because our main objective in this paper was to include the proof system for FOL with equality as part of our proof system, to indicate that conventional reasoning remains valid and thus automated provers can be used unchanged. It is likely, however, that a minimal proof system working directly with the definedness symbols $\lceil \_ \rceil$ can be obtained such that the equality and membership axioms and rules in Figure 5 can be proved as lemmas.

Our completeness result in Section 11 relies heavily on equality and on membership patterns, whose definitions require the existence of the definedness symbols $\lceil \_ \rceil$. On the other hand, Proposition 10.1 translates arbitrary matching logic validity to validity in predicate logic with equality, even when there are no definedness symbols. Since predicate logic with equality admits complete deduction, we conjecture that matching logic must admit an alternative complete proof system which does not rely on definedness symbols.

We have not discussed any computationally effective fragments of matching logic or heuristics to automate matching logic deduction. These are crucial for the development of practical provers and program verifiers. The systematic study of such fragments and heuristics is left for future work. Also, complexity results in the style of [21, 1, 16, 51] for separation logic can likely also be obtained for fragments of matching logic.

Many of the results related to localizing/globalizing reasoning, such as Propositions 2.10, 5.12, and 2.11, extend to monotone/positive contexts, that is, to ones without negations on the path to the placeholder. While non-monotonic contexts do not seem to occur frequently in program verification efforts, it would nevertheless be worthwhile investigating techniques for the elimination of negation, likely generalizing those in [58, 88, 36], or intuitionistic variants of matching logic where negation is not allowed at all in patterns.

Finally, the main application of matching logic so far was as a pattern language for reachability logic [28, 27, 80, 85], where reachability rules, which are pairs of patterns $\varphi \Rightarrow \varphi'$, can be used to specify both operational semantics rules and properties to prove about programs. Reachability logic has its own (language independent) sound and relatively complete proof system. We conjecture that we can capture reachability logic as an instance of matching logic, too, in a similar vein to how we did it for modal logic in Section 8: add some new symbols with their (axiomatized) semantics and then prove the proof rules of reachability logic as lemmas/corollaries. For example, we can extend the world models M in Section 8

with a Kripke transition relation $w\,R\,w'$ by adding a symbol $\circ\_ \in \Sigma_{World,World}$ and assuming $w\,R\,w'$ iff $w \in \circ_M(w')$, then define $\Diamond$ and other CTL or even CTL* operators as least fixed points, and finally the reachability rules as sugar.

Acknowledgments. This is an extended version of an RTA'15 invited paper [79]. The author warmly thanks the RTA'15 program committee for the invitation and to the anonymous reviewers. The author also expresses his deepest thanks to the $\mathbb{K}$ team (`http://kframework.org`), who share the belief that programming languages should have only one semantics, which should be executable, and formal analysis tools, including fully fledged deductive program verifiers, should be obtained from such semantics at little or no extra cost. I would like to also warmly thank the following colleagues and friends for their comments and criticisms on previous drafts of this paper: Nikolaj Bjorner, Xiaohong Chen, Claudia-Elena Chiriţă, Maribel Fernández, Ioana Leuştean, Dorel Lucanu, José Meseguer, Brandon Moore, Daejun Park, Cosmin Rădoi, Traian Florin Şerbănuţă, and Andrei Ştefănescu.

References

[1] T. Antonopoulos, N. Gorogiannis, C. Haase, M. I. Kanovich, and J. Ouaknine. Foundations for decision problems in separation logic with general inductive predicates. In *FOSSACS'14*, volume 8412 of *LNCS*, pages 411–425, 2014.

[2] A. Appel and S. Blazy. Separation logic for small-step Cminor. In *TPHOLs'07*, volume 4732 of *LNCS*, pages 5–21, 2007.

[3] A. W. Appel. Verified software toolchain. In *ESOP'11*, volume 6602 of *LNCS*, pages 1–17, 2011.

[4] M. Barnett, B. yuh Evan Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO'05*, volume 4111 of *LNCS*, pages 364–387, 2006.

[5] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In *CAV'11*, volume 6806 of *LNCS*, pages 171–177. Springer, 2011.

[6] C. Barrett and C. Tinelli. CVC3. In *CAV'07*, volume 4590 of *LNCS*, pages 298–302, 2007.

[7] O. Becker. Zur logik der modalitäten. *Jahrbuch für Philosophie und Phänomenologische Forschung*, 11:497–548, 1930.

[8] J. Berdine, C. Calcagno, and P. O'Hearn. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *FMCO'05*, volume 4111 of *LNCS*, pages 115–137, 2005.

[9] J. Berdine, B. Cook, and S. Ishtiaq. SLAyer: Memory safety for systems-level code. In *CAV'11*, volume 6806 of *LNCS*, pages 178–183, 2011.

[10] D. Beyer. Reliable and reproducible competition results with benchexec and witnesses (report on SV-COMP 2016). In *TACAS'16*, volume 9636 of *LNCS*, pages 887–904, 2016.

[11] P. Blackburn, M. de Rijke, and Y. Venema. *Modal Logic*. Cambridge University Press, NY, USA, 2001.

[12] F. Bobot and J.-C. Filliâtre. Separation predicates: A taste of separation logic in first-order logic. In *ICFEM'12*, volume 7635 of *LNCS*, pages 167–181, 2012.

[13] M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith. A trusted mechanised JavaScript specification. In *POPL'14*, pages 87–100. ACM, 2014.

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

[15] M. Botincan, M. Parkinson, and W. Schulte. Separation Logic Verification of C Programs with an SMT Solver. *Electronic Notes in Theoretical Computer Science*, 254:5–23, 2009.

[16] J. Brotherston, C. Fuhs, J. A. N. Pérez, and N. Gorogiannis. A decision procedure for satisfiability in separation logic with inductive predicates. In *CSL-LICS'14*, pages 25:1–25:10. ACM, 2014.

[17] J. Brotherston, N. Gorogiannis, M. Kanovich, and R. Rowe. Model Checking for Symbolic-Heap Separation Logic with Inductive Predicates. In *POPL'16*, pages 84–96. ACM, 2016.

[18] J. Brotherston and J. Villard. Parametric Completeness for Separation Theories. In *POPL'14*, pages 453–464. ACM, 2014.

[19] R. M. Burstall and J. A. Goguen. Putting theories together to make specifications. In *IJCAI'77*, pages 1045–1058. Morgan Kaufmann Publishers Inc., 1977.

[20] C. Calcagno, P. Gardner, and M. Hague. From separation logic to first-order logic. In *FOSSACS'05*, volume 3441 of *LNCS*, pages 395–409, 2005.
[21] C. Calcagno, H. Yang, and P. W. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In *FSTTCS'01*, volume 2245 of *LNCS*, pages 108–119, 2001.
[22] C. Cao, J. Wang, Y. Sui, and Y. Shen. Translating separation logic into a fragment of the first-order logic. *International Conference on Semantics, Knowledge and Grid*, pages 188–194, 2010.
[23] A. Cholewa, S. Escobar, and J. Meseguer. Constrained narrowing for conditional equational theories modulo axioms. *Science of Computer Programming*, 112:24–57, 2015.
[24] D.-H. Chu, J. Jaffar, and M.-T. Trinh. Automatic Induction Proofs of Data-Structures in Imperative Programs. In *PLDI'15*, pages 457–466. ACM, 2015.
[25] M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, and C. Talcott. *All About Maude*, volume 4350 of *LNCS*. Springer, 2007.
[26] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs'09*, volume 5674 of *LNCS*, pages 23–42, 2009.
[27] 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.
[28] 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.
[29] L. De Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS'08*, volume 4963 of *LNCS*, pages 337–340, 2008.
[30] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340, 2008.
[31] R. Diaconescu and K. Futatsugi. *CafeOBJ Report. The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*, volume 6 of *AMAST Series in Computing*. World Scientific, 1998.
[32] D. Distefano and M. J. Parkinson. jStar: Towards practical verification for Java. In *OOPSLA'08*, pages 213–226. ACM, 2008.
[33] B. Dutertre. Yices 2.2. In *CAV'14*, volume 8559 of *LNCS*, pages 737–744, 2014.
[34] C. Ellison and G. Rosu. An executable formal semantics of C with applications. In *POPL*, pages 533–544. ACM, 2012.
[35] W. M. Farmer and J. D. Guttman. A set theory with support for partial functions. *Studia Logica*, 66(1):59–78, 2000.
[36] M. Fernández. Negation elimination in empty or permutative theories. *Journal of Symbolic Computation*, 26(1):97–133, 1998.
[37] D. Filaretti and S. Maffeis. An executable formal semantics of php. In *ECOOP'14*, LNCS, pages 567–592. Springer, 2014.
[38] J. Filliâtre and A. Paskevich. Why3 - where programs meet provers. In *ESOP'13*, volume 7792 of *LNCS*, pages 125–128, 2013.
[39] K. Gödel. Die vollständigkeit der axiome des logischen funktionenkalküls. *Monatshefte für Mathematik und Physik*, 37(1):349–360, 1930.
[40] K. Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme I. *Monatshefte für Mathematik und Physik*, 38(1):173–198, 1931.
[41] J. Goguen, J. Thatcher, and E. Wagner. *An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types*. IBM research reports. IBM Research Center, 1976.
[42] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In *Software Engineering with OBJ: Algebraic specification in action*, pages 3–167. Kluwer, 2000.
[43] 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.
[44] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebras. *Journal of the Association for Computing Machinery*, 24(1):68–95, Jan. 1977.
[45] R. Goldblatt. Mathematical modal logic: A view of its evolution. *Journal of Applied Logic*, 1(5-6):309–392, 2003.
[46] 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`.

[47] D. Harel, D. Kozen, and J. Tiuryn. Dynamic logic. In *Handbook of Philosophical Logic*, pages 497–604, 1984.
[48] J. Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, New York, NY, USA, 1st edition, 2009.
[49] C. Hathhorn, C. Ellison, and G. Roşu. Defining the undefinedness of C. In *PLDI'15*, pages 336–345. ACM, 2015.
[50] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the Association for Computing Machinery*, 12(10):576–580, 1969.
[51] R. Iosif, A. Rogalewicz, and J. Simácek. The tree width of separation logic with recursive definitions. In *CADE'13*, volume 7898 of *LNCS*, 2013.
[52] B. Jacobs. Weakest pre-condition reasoning for Java programs with JML annotations. *The Journal of Logic and Algebraic Programming*, 58(1-2):61–88, 2004.
[53] C. B. Jay. The pattern calculus. *ACM Transactions on Programming Languages and Systems*, 26(6):911–937, 2004.
[54] C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, J. A. McCarthy, J. Rafkind, S. Tobin-Hochstadt, and R. B. Findler. Run your research: On the effectiveness of lightweight mechanization. In *POPL'12*, pages 285–296. ACM, 2012.
[55] R. Krebbers, A. Timany, and L. Birkedal. Interactive Proofs in Higher-Order Concurrent Separation Logic. In *POPL'17*, pages 457–466. ACM, 2017.
[56] G. Kreisel and J. L. Krivine. *Elements of mathematical logic (Model theory)*. North Holland Publishing Company, Amsterdam, 1967.
[57] S. A. Kripke. A completeness theorem in modal logic. *Journal of Symbolic Logic*, 24(1):1–14, 03 1959.
[58] J. Lassez, M. J. Maher, and K. Marriott. Elimination of negation in term algebras. In *MFCS'91*, volume 520 of *LNCS*, pages 1–16, 1991.
[59] J. Lassez and K. Marriott. Explicit representation of terms defined by counter examples. *Journal of Automated Reasoning*, 3(3):301–317, 1987.
[60] B. Liskov and S. Zilles. Programming with abstract data types. *SIGPLAN Not.*, 9(4):50–59, Mar. 1974.
[61] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0, `http://coq.inria.fr`.
[62] J. Meseguer and S. Skeirik. Equational formulas and pattern operations in initial order-sorted algebras. In *LOPSTR'15*, volume 9527 of *LNCS*, pages 36–53. Springer, 2015.
[63] P. D. Mosses. *CASL Reference Manual, The Complete Documentation of the Common Algebraic Specification Language*, volume 2960 of *LNCS*. Springer, 2004.
[64] J. A. Navarro Perez and A. Rybalchenko. Separation logic + superposition calculus = heap theorem prover. In *PLDI'11*, pages 556–566. ACM, 2011.
[65] J. A. Navarro Perez and A. Rybalchenko. Separation logic modulo theories. In *APLAS'13*, volume 8301 of *LNCS*, pages 90–106, 2013.
[66] T. Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10:171–186, 1998.
[67] T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002.
[68] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL'01*, volume 2142 of *LNCS*, pages 1–19, 2001.
[69] P. W. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.
[70] S. Owens. A sound semantics for OCaml$_{light}$. In *ESOP'08*, volume 4960 of *LNCS*, pages 1–15, 2008.
[71] D. Park, A. Ştefănescu, and G. Roşu. KJS: A complete formal semantics of JavaScript. In *PLDI'15*, pages 346–356. ACM, 2015.
[72] M. J. Parkinson and A. J. Summers. The relationship between separation logic and implicit dynamic frames. In *ESOP'11*, volume 6602 of *LNCS*, pages 439–458, 2011.
[73] E. Pek, X. Qiu, and P. Madhusudan. Natural Proofs for Data Structure Manipulation in C using Separation Logic. In *PLDI'14*, pages 440–451. ACM, 2014.
[74] R. Pichler. Explicit versus implicit representations of subsets of the Herbrand universe. *Theoretical Computer Science*, 290(1):1021–1056, 2003.

[75] R. Piskac, T. Wies, and D. Zufferey. Automating separation logic using smt. In *CAV'13*, volume 8044 of *LNCS*, pages 773–789, 2013.
[76] G. D. Plotkin. A note on inductive generalization. *Machine Intelligence*, 5:153–163, 1970.
[77] 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.
[78] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS'02*, pages 55–74. IEEE, 2002.
[79] G. Roşu. Matching logic – extended abstract. In *RTA'15*, volume 36 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5–21, July 2015.
[80] G. Roşu, A. Ştefănescu, Ştefan Ciobâcă, and B. M. Moore. One-path reachability logic. In *LICS'13*, pages 358–367. IEEE, 2013.
[81] G. Rosu, C. Ellison, and W. Schulte. Matching logic: An alternative to Hoare/Floyd logic. In *AMAST'10*, volume 6486 of *LNCS*, pages 142–162, 2010.
[82] G. Roşu and T. F. Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.
[83] G. Rosu and T. F. Serbanuta. K overview and simple case study. In *Proceedings of International K Workshop (K'11)*, volume 304 of *ENTCS*, pages 3–56. Elsevier, June 2014.
[84] G. Rosu and A. Stefanescu. Matching logic: a new program verification approach (NIER track). In *ICSE-NIER'11*, pages 868–871. ACM, 2011.
[85] G. Rosu and A. Stefanescu. Checking reachability using matching logic. In *OOPSLA'12*, pages 555–574. ACM, 2012.
[86] G. Rosu and A. Stefanescu. From hoare logic to matching logic reachability. In *FM'12*, volume 7436 of *LNCS*, pages 387–402, 2012.
[87] P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strnisa. Ott: effective tool support for the working semanticist. In *ICFP'07*, pages 1–12. ACM, 2007.
[88] M. Tajine. The negation elimination from syntactic equational formula is decidable. In *RTA'93*, volume 690 of *LNCS*, pages 316–327, 1993.
[89] A. Tarski and S. Givant. A formalization of set theory without variables. *Colloquium Publications*, 41, 1987.
[90] J. F. A. K. van Benthem. *Modal Logic and Classical Logic*. Bibliopolis, Naples. Distributed in the U.S.A. By Humanities Press, 1983.
[91] Wikipedia. Abstract data type, 2016. `https://en.wikipedia.org/wiki/Abstract_data_type`.

$\mathbb{K}$ implements reachability logic, the same way Maude implements rewriting logic. Since patterns allow variables and constraints on them in configurations, $\mathbb{K}$ rewriting becomes symbolic execution with the semantic rules of the language. Its symbolic execution engine is connected to the Z3 SMT solver [30]. We next show an example C program verified with our current implementation of reachability logic in $\mathbb{K}$, mentioning that we have similarly verified various programs manipulating lists and trees, performing arithmetic and I/O operations, and implementing sorting algorithms, binary search trees, AVL trees, and the Schorr-Waite graph marking algorithm. The Matching Logic web page, `http://matching-logic.org`, contains an online interface to run MatchC, an instance of our verifier for C, where users can try more than 50 existing examples (or upload their own). To simplify writing properties, MatchC allows users to write reachability rules and invariant patterns as comments in the C program.

Figure 6 shows the classic list reverse program, together with all the specifications that the user of MatchC has to provide (grayed areas, given as code annotations). MatchC verifies this program for full correctness, not only memory safety, in 0.06 seconds. The user-provided specifications are translated into reachability rule proof obligations by MatchC and then attempted to be proved automatically. The "$" stands for the function body, the "..." for structural frame variables, the variables starting with "?" are existentially quantified over the current formula, etc. We do not mean to explain the MatchC notation in detail here; we only show this example to highlight the fact that reachability logic verification, in

```
struct listNode { int val; struct listNode *next; };
struct listNode* reverseList(struct listNode *x)
```
rule $\langle \$ \Rightarrow \texttt{return}\ ?\mathsf{p};\ \cdots \rangle_{\mathsf{k}}\ \langle \cdots\ \mathit{heap}\ \cdots \rangle \mathsf{list(x)(A)} \Rightarrow \mathsf{list}(?\mathsf{p})(\mathsf{rev}(\mathsf{A}))$

```
{
  struct listNode *p, *y;
  p = NULL;
```
inv $\langle \cdots\ \mathit{heap}\ \cdots \rangle \mathsf{list(p)}(?\mathsf{B}),\ \mathsf{list(x)}(?\mathsf{C})\ \wedge\ \mathsf{A} = \mathsf{rev}(?\mathsf{B})@?\mathsf{C}$

```
  while(x != NULL) {
    y = x->next;
    x->next = p;
    p = x;
    x = y;
  }
  return p;
}
```

Figure 6: C function reversing a singly-linked list.

spite of being based on "low-level" operational semantics, still allows a comfortable level of abstraction.

Let $\mathcal{A}$ be the rewrite system giving the semantics of the C language, and let $\mathcal{C}$ be the set of reachability rules corresponding to user-provided specifications (properties that one wants to verify, like the grayed ones above). MatchC derives the rules in $\mathcal{C}$ using the proof system in Figure 5. It begins by applying Circularity for each rule in $\mathcal{C}$ and reduces the task to deriving individual sequents of the form $\mathcal{A} \vdash_{\mathcal{C}} \varphi \Rightarrow \varphi'$. To prove them, MatchC rewrites φ using rules in $\mathcal{A} \cup \mathcal{C}$ searching for a formula that implies φ'. Whenever the semantics rule for `if` in $\mathcal{A}$ cannot apply because its condition is symbolic, a Case Analysis is applied and formula split into a disjunction. When no rule can be applied, abstraction axioms are attempted. If application of an abstraction axiom would result into a more concrete formula, the verifier applies the respective axiom (for instance, knowing the head of a linked list is not null results in an automatic list unrolling).

Regarding from reachability logic's perspective, $\mathbb{K}$ consists of a collection of heuristics and optimizations to perform *proof search*, that is, to derive proof derivations using the reachability logic proof system. For example, suppose that the initial configuration pattern is all concrete/ground (i.e., when it contains no variables) and that $\mathbb{K}$ is requested to use its rewrite engine to simply execute the program in its initial state. In terms of proof derivation with the one-path reachability proof system in Figure 5 and its more general variant in [80], this corresponds to a derivation of a one-path reachability rule. If $\mathbb{K}$ is requested to search the entire configuration-space to find all the configurations that can be reached as a consequence of a non-deterministic semantics, that corresponds to a derivation of an all-path reachability rule using the generalized proof system in [27]. Similarly, when using $\mathbb{K}$'s model-checking capabilities or when deductively verifying programs like above,

all we do can be framed in terms of deriving proofs using a rigorously defined sound and relatively complete proof system.

13.1. **Structural Framing.** Heap framing is a major outcome of the use of separation logic for program verification, since it enables local reasoning:

$$\vdash \{\varphi_{pre}\}\,\mathsf{c}\,\{\varphi_{post}\} \text{ implies}^4 \vdash \{\varphi_{pre} * \varphi\}\,\mathsf{c}\,\{\varphi_{post} * \varphi\}$$

Matching logic can be identically used instead of separation logic in axiomatic semantics. In particular, the heap framing rule above would stay unchanged. However, if used in combination with reachability logic [80, 27], matching logic enables us to develop more flexible and more general types of framing. Regarding flexibility, note that we may not always want to automatically assume heap framing (e.g., when memory is finite—embedded systems, device drivers, etc.—, or when the language has functions like getTotalMemory() returning the available memory). Regarding generality, we may want similar framing rules for other semantic cells in the program configuration, such as input/output buffers, exception stacks, thread resources, etc.

Consider the (semantic) rewrite rule of assignment in a C-like language, whose configuration contains an environment map from variables to locations and a heap map from locations to values (say due to the "address" construct &):

$$\begin{array}{rl} & \langle\langle \mathsf{x} = \mathsf{v}; R\rangle_{\mathsf{k}}\ \langle \mathsf{x} \mapsto l * e\rangle_{\mathsf{env}}\ \langle l \mapsto \_ * h\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{cfg}} \\ \Rightarrow & \langle\langle\qquad R\rangle_{\mathsf{k}}\ \langle \mathsf{x} \mapsto l * e\rangle_{\mathsf{env}}\ \langle l \mapsto \mathsf{v} * h\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{cfg}} \end{array}$$

The variables R, e, h and c can all be thought of as *structural frames*: R is the code frame, e is the environment frame, h is the heap frame, and c is the configuration frame. The assignment rule above says that the value at the location l of x in the heap changes to v regardless of what the structural frames match.

The same specification style extends to arbitrary reachability properties, without a need to define an axiomatic semantics. For example, the (pre-/post-condition) specification of the function in Fig. 2 can be stated as the following rule between patterns:

$$\begin{array}{l} \langle\langle \mathsf{body}; R\rangle_{\mathsf{k}} \langle \mathsf{n} \mapsto l * e\rangle_{\mathsf{env}} \langle l \mapsto n * h\rangle_{\mathsf{heap}} \langle \mathsf{A}, I\rangle_{\mathsf{in}} \langle O\rangle_{\mathsf{out}}\ c\rangle_{\mathsf{cfg}} \\ \quad \wedge\ n = \mathsf{len}(\mathsf{A}) \\ \Rightarrow \exists n'. \langle\langle R\rangle_{\mathsf{k}} \langle \mathsf{n} \mapsto l * e\rangle_{\mathsf{env}} \langle l \mapsto n' * h\rangle_{\mathsf{heap}} \langle I\rangle_{\mathsf{in}} \langle O, \mathsf{rev}(\mathsf{A})\rangle_{\mathsf{out}}\ c\rangle_{\mathsf{cfg}} \end{array}$$

If we want to also state that n is not modified, then we remove the existential quantifier and replace n' with n. If we want to say, for whatever reason, that the heap must be empty when this function is invoked, then we remove the heap frame h. If we want to state that the size of the available memory must be larger than a certain limit, then we add the constraint $\mathsf{size}(h) \geq \mathit{limit}$ to the LHS pattern. Similarly for the other structural frames, O, I, and c. It should be clear that this gives us significant power in what kind of properties we can specify. Reachability logic [80, 27] provides a language-independent sound and relatively complete proof system to derive such reachability rules, starting with the formal operational semantics of the language.

If a language semantics is so that structural framing in a particular semantic cell is always sound, say in the $\langle ... \rangle_{\mathsf{heap}}$ cell, then one can prove a property "$\langle\langle h_1\rangle_{\mathsf{heap}}\ c_1\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle h_2\rangle_{\mathsf{heap}}\ c_2\rangle_{\mathsf{cfg}}$ implies $\langle\langle h_1 * h\rangle_{\mathsf{heap}}\ c_1\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle h_2 * h\rangle_{\mathsf{heap}}\ c_2\rangle_{\mathsf{cfg}}$ when side-condition". Such a property can be proved, e.g., when all semantic rules use an unconstrained heap frame h (like

[4]Depending on the language, the rule may also have side conditions on the locations accessed by c and φ, but those are irrelevant for our discussion here.

in the assignment rule above). However, such a rule would not hold in a language providing, e.g., a construct that returns the size of the available memory. It is worth mentioning that although possible, such structural framing rules are unnecessary. That is because the structural frames are plain first-order variables that obey the general pattern matching principles like the other variables, so nothing special needs to be done about them. Indeed, in MatchC, the only difference between a framed and an unframed variant of a property is the use of "...".

14. Others

14.1. **Binders.**