AMAST'10, LNCS 6486, pp 142-162. 2010

Matching Logic: An Alternative to Hoare/Floyd Logic[⋆]

Grigore Roşu[1], Chucky Ellison[1], and Wolfram Schulte[2]

[1] University of Illinois at Urbana-Champaign {grosu, celliso2}@illinois.edu
[2] Microsoft Research, Redmond schulte@microsoft.com

Abstract. This paper introduces matching logic, a novel framework for defining axiomatic semantics for programming languages, inspired from operational semantics. Matching logic specifications are particular first-order formulae with constrained algebraic structure, called *patterns*. Program configurations satisfy patterns iff they *match* their algebraic structure and satisfy their constraints. Using a simple imperative language (IMP), it is shown that a restricted use of the matching logic proof system is equivalent to IMP's Hoare logic proof system, in that any proof derived using either can be turned into a proof using the other. Extensions to IMP including a heap with dynamic memory allocation and pointer arithmetic are given, requiring no extension of the underlying first-order logic; moreover, heap patterns such as lists, trees, queues, graphs, etc., are given algebraically using fist-order constraints over patterns.

1 Introduction

Hoare logic, often identified with axiomatic semantics, was proposed more than forty years ago [15] as a rigorous means to reason about program correctness. A Hoare logic for a language is given as a formal system deriving *Hoare triples* of the form $\{\varphi\}$ s $\{\varphi'\}$, where s is a statement and φ and φ' are state properties expressed as logical formulae, called *precondition* and *postcondition*, respectively. Most of the rules in a Hoare logic proof system are language-specific. Programs and state properties in Hoare logic are connected by means of program variables, in that properties φ, φ' in Hoare triples $\{\varphi\}$ s $\{\varphi'\}$ can and typically do refer to program variables that appear in s. Moreover, Hoare logic assumes that the expression constructs of the programming language are also included in the logical formalism used for specifying properties, typically first-order logic (FOL). Hoare logic is deliberately abstract, in that it is not concerned with "low-level" operational aspects, such as how the program state is represented or how the program is executed.

In spite of serving as the foundation for many program verification tools and frameworks, it is well-known that it is difficult to specify and prove properties about the heap (i.e., dynamically allocated, shared mutable objects) in Hoare logic. In particular, local reasoning is difficult because of the difficulty of frame inference [23, 25, 29]. Also, it is difficult to specify recursive predicates, because they raise both theoretical (consistency) and practical (hard to automatically reason with) concerns. Solutions are either

⋆ Supported in part by NSF grants CCF-0916893, CNS-0720512, and CCF-0448501, by NASA contract NNL08AA23C, by a Samsung SAIT grant, and by several Microsoft gifts.

ad hoc [18], or involve changing logics [21]. Finally, program verifiers based on Hoare logic often yield proofs which are hard to debug and understand, because these typically make heavy use of encoding (of the various program state components into a flat FOL formula) and follow a backwards verification approach (based on weakest precondition).

Separation logic takes the heap as its central concern and attempts to address the limitations above by *extending* Hoare logic with special logical connectives, such as the separating conjunct "$*$" [23, 25], allowing one to specify properties that hold in disjoint portions of the heap. The axiomatic semantics of heap constructs can be given in a forwards manner using separation connectives. While program verification based on separation logic is an elegant approach, it is unfortunate that one would need to extend the underlying logic, and in particular theorem provers, to address new language features.

In an effort to overcome the limitations above and to narrow the gap between operational semantics (easy) and program verification (hard), we introduce *matching logic*, which is designed to be agnostic with respect to the underlying language configuration, as long as it can be expressed in a standard algebraic way. Matching logic is similar to Hoare logic in many aspects. Like Hoare logic, matching logic specifies program states as logical formulae and gives axiomatic semantics to programming languages in terms of pre- and post-conditions. Like Hoare logic, matching logic can generically be extended to a formal, syntax-oriented compositional proof system. However, unlike Hoare logic, matching logic specifications are not flattened to arbitrary FOL formulas. Instead, they are kept as symbolic configurations, or *patterns*, i.e., restricted $\mathrm{FOL}_=$ (FOL with equality) formulae possibly containing free and bound (existentially quantified) variables. This allows the logic to stay the same for different languages—new symbols can simply be added to the signature, together with new axioms defining their behavior.

Matching Logic Patterns. Patterns (Sec. 3) can be defined on top of any algebraic specification of configurations. In this paper, the simple language IMP (Sec. 2) uses two-item configurations $\langle\langle\ldots\rangle_k\ \langle\ldots\rangle_{env}\rangle$, where $\langle\ldots\rangle_k$ holds a fragment of program and $\langle\ldots\rangle_{env}$ holds an environment (map from program variables to values). The order of items in the configuration does not matter, i.e., the top $\langle\ldots\rangle$ cell holds a set. We use the `typewriter` font for code and *italic* for logical variables. A possible configuration γ is:

$$\langle\langle \texttt{x := 1; y := 2}\rangle_k\ \langle \texttt{x} \mapsto 3,\ \texttt{y} \mapsto 3,\ \texttt{z} \mapsto 5\rangle_{env}\rangle$$

One pattern p that is *matched* by the above configuration γ is the $\mathrm{FOL}_=$ formula:

$$\exists a, \rho.((\square = \langle\langle \texttt{x := 1; y := 2}\rangle_k\ \langle \texttt{x} \mapsto a,\ \texttt{y} \mapsto a,\ \rho\rangle_{env}\rangle) \wedge a \geq 0)$$

where "$\square$" is a placeholder for configurations (regarded as a special variable). To see that γ matches p, we replace $\square$ with γ and prove the resulting $\mathrm{FOL}_=$ formula (bind ρ to "$\texttt{z} \mapsto 5$" and a to 3). For uniformity, we prefer to use the notation (described in Sec. 3)

$$\langle\langle \texttt{x := 1; y := 2}\rangle_k\ \langle \texttt{x} \mapsto ?a,\ \texttt{y} \mapsto ?a,\ ?\rho\rangle_{env}\ \langle ?a \geq 0\rangle_{form}\rangle$$

for patterns, which includes pattern's *constraints* as an item $\langle\ldots\rangle_{form}$ in its *structure* $\langle\ldots\rangle$, at the same time eliminating the declaration of locally bound variables but tagging them with ? to distinguish them from other variables; we also allow free variables in patterns, acting as proof parameters (like in Hoare logic). It is this notation for patterns that inspired us to call their satisfaction *matching*, which justifies the name of our logic.

Extending IMP with a heap gives us the language HIMP (Sec. 5), which uses configurations of the form $\langle\langle\ldots\rangle_k\ \langle\ldots\rangle_{env}\ \langle\ldots\rangle_{mem}\rangle$; one possible HIMP configuration is:

$$\langle\langle \texttt{[x] := 5; z := [y]}\rangle_k\ \langle \texttt{x} \mapsto 2,\ \texttt{y} \mapsto 2\rangle_{env}\ \langle 2 \mapsto 7\rangle_{mem}\rangle$$

describing a program that will first assign 5 to the location pointed to by `x`, then assign the value pointed to by `y` to `z`. Note that pointers `x` and `y` are aliased here, so `z` will receive 5. One of many patterns that are matched by the above configuration is:

$$\langle\langle \texttt{[x] := 5; z := [y]}\rangle_k\ \langle \texttt{x} \mapsto ?a,\ \texttt{y} \mapsto ?a,\ ?\rho\rangle_{env}\ \langle ?a \mapsto ?v,\ ?\sigma\rangle_{mem}\ \langle ?a \geq 0\rangle_{form}\rangle$$

The above pattern specifies all the configurations where `x` and `y` are allocated and aliased. Matching logic can therefore express separation at the term level instead of at the formula level; in particular, no disjointness predicate is needed (see Sec. 5 and 6). Note that the constraint $?a \geq 0$ is redundant in the pattern above, because one can infer it from the fact that the binding $?a \mapsto ?v$ appears in $\langle\ldots\rangle_{mem}$ (the cell $\langle\ldots\rangle_{mem}$ wraps a map structure whose domain is the non-negative integers). To simplify the presentation, in general we assume that the $\langle\ldots\rangle_{form}$ cell appearing in a pattern includes all the current constraints of the pattern, i.e., not only those related to the program itself (like the $?a \geq 0$ on the previous page) but also those pertaining to or resulting from the background mathematical theories; in our implementation, for practical reasons we actually keep the two distinct and only derive consequences of the background theories (like the $?a \geq 0$ above) by need.

Assignment in Matching vs. Hoare/Floyd Logic. Both Hoare logic and matching logic are defined as language-specific proof systems, the former deriving triples $\{\varphi\}\ \texttt{s}\ \{\varphi'\}$ as explained and the latter deriving pairs of patterns $\Gamma \Downarrow \Gamma'$, with the intuition that if program configuration (which includes the code) γ matches Γ then the configuration γ' obtained after completely reducing γ matches Γ'; we only discuss partial correctness in this paper. To highlight a fundamental difference between Hoare logic and matching logic, let us examine the assignment rules for each. The Hoare logic assignment rule,

$$\{\varphi[\texttt{e}/\texttt{x}]\}\ \texttt{x := e}\ \{\varphi\}$$

or (HL-asgn) in Fig. 1, is perhaps the most representative rule of Hoare logic, showing how program variables and expressions are mixed in formulae: if the program state satisfies the formula $\varphi[\texttt{e}/\texttt{x}]$ (i.e., φ in which all free occurrences of variable `x` are replaced by expression `e`) before the assignment statement "`x := e`", then the program state will satisfy φ after the execution of the assignment statement (IMP has side-effect-free expressions).

In contrast, the matching logic rule for assignment,

$$\frac{\langle\langle \texttt{e}\rangle_k\, C\rangle \Downarrow \langle\langle v\rangle_k\, C\rangle}{\langle\langle \texttt{x := e}\rangle_k\, C\rangle \Downarrow \langle\langle \cdot\rangle_k\, C[\texttt{x} \leftarrow v]\rangle}$$

or (ML-asgn) in Fig. 2, says that if `e` reduces to v in a program configuration matching pattern $\langle\langle \texttt{e}\rangle_k\, C\rangle$ (for IMP, C only contains an environment item; however, we prefer to use a meta-variable C to increase the modularity of our proof system), then after executing the assignment "`x := e`" the configuration matches $\langle\langle\cdot\rangle_k\, C[\mathtt{x} \leftarrow v]\rangle$, where the "$\cdot$" stands for the empty or completed computation and where $C[\mathtt{x} \leftarrow v]$ is some operation in the algebraic data type of configurations which updates the binding of `x` to v in C's environment; in the case of IMP, for example, this latter operation can be defined using an equation $\langle\rho\rangle_{env}[\mathtt{x} \leftarrow v] = \langle\rho[v/\mathtt{x}]\rangle_{env}$ (we assume the map update operation _[_/_] already defined as part of the mathematical background theory). The most apparent difference is the forward proof style used. The proof proceeds in the order of execution, and the environment is explicit (as an item in C), instead of encoded in a flat formula, like in Hoare logic. In fact, replacing the patterns with actual configurations that match them gives a rule that would not be out of place in standard big-step operational semantics.

Even though the matching rule above is a forward rule, it is not the Floyd rule [10]:

$$\{\varphi\}\ \mathtt{x} := \mathtt{e}\ \{\exists v.\,(\mathtt{x} = \mathtt{e}[v/\mathtt{x}]) \wedge \varphi[v\,/\,\mathtt{x}]\}$$

The Floyd rule above is indeed forward, but the reader should notice that its use results in the addition of an existential quantifier. For a given program, this rule might result in the introduction of many quantifiers, which are often hard for verification tools to handle tractably. Looked at in this light, the matching logic rule offers the best of both worlds—the rule is both forwards and does not introduce quantifiers. Indeed, *none* of the matching logic proof rules introduce quantifiers. Working forward is arguably more "natural". Additionally, as discussed in [12], working forward has a number of other concrete advantages related to reverse engineering, testing, debugging, model-checking, etc.

Contributions. The main contribution of this paper is the formal introduction of *matching logic*. We show that for the simple imperative language IMP, a restricted use of the matching logic proof system is equivalent to a conventional Hoare logic proof system. The translation from Hoare to matching logic is generic and should work for any language, suggesting that any Hoare logic proof system admits an equivalent matching logic proof system. However, the other translation, going from matching logic to Hoare logic, appears to be language specific because it relies on finding appropriate encodings of program configuration patterns into Hoare specifications; it is not clear that they always exist.

Section 2 discusses preliminaries and IMP. Secion 3 describes the details of matching logic and gives a matching logic semantics to IMP. Using IMP, Section 4 establishes the relationship between matching logic and Hoare logic. Section 5 shows how easily the concepts and techniques extend to a language with a heap. Finally, Sec. 6 discusses an example and our experience with a matching logic verifier implemented in Maude.

2 Preliminaries, the IMP Language, and Hoare Logic

Here we introduce material necessary to understand the contributions and results of this paper. First we discuss background material, then the configuration and notations we use for IMP, and finally Hoare logic by means of IMP.

Preliminaries. We assume the reader is familiar with basic concepts of algebraic specification and first-order logic with equality. The role of this section is to establish our notation for concepts and notions used later in the paper. An *algebraic signature* $(\mathbb{S}, \Sigma)$ is a finite set of *sorts* $\mathbb{S}$ and a finite set of *operation symbols* Σ over sorts in $\mathbb{S}$. For example, $\mathbb{S}$ may include sorts E for expressions and S for statements, and Σ may include operation symbols like "`if(_)_else_` : $E \times S \times S \to S$". Here we used the mixfix notation for operation symbols, where underscores are placeholders for arguments, which is equivalent to the context-free or BNF notation. Since the latter is more common for defining programming language syntax, we prefer it from here on; so we write "S ::= $\mathtt{if}\,(E)\,S\,\mathtt{else}\,S$". We write Σ instead of $(\mathbb{S}, \Sigma)$ when $\mathbb{S}$ is understood or irrelevant. We let $\mathcal{T}_\Sigma$ denote the *initial Σ-algebra of ground terms* (i.e., terms without variables) and $\mathcal{T}_\Sigma(X)$ denote the *free Σ-algebra of terms with variables in X*, where X is an $\mathbb{S}$-indexed set of variables.

We next briefly recall *first-order logic with equality (FOL$_=$)*. A *first-order signature* $(\mathbb{S}, \Sigma, \Pi)$ extends an algebraic signature $(\mathbb{S}, \Sigma)$ with a finite set of predicate symbols Π. FOL$_=$ formulae have the syntax $\varphi ::= t = t' \mid \pi(\bar{t}) \mid \exists X.(\varphi) \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2$, plus the usual derived constructs $\varphi_1 \vee \varphi_2$, $\varphi_1 \Rightarrow \varphi_2$, $\forall X.(\varphi)$, etc., where t, t' range over Σ-terms of the same sort, $\pi \in \Pi$ over atomic predicates, $\bar{t}$ over appropriate term tuples, and X over finite sets of variables. Σ-terms can have variables; all variables are chosen from a fixed sort-wise infinite $\mathbb{S}$-indexed set of variables, *Var*. We adopt the notation $\varphi[\mathtt{e}/\mathtt{x}]$ for the capture-free substitution of all free occurrences of variable $\mathtt{x}$ by term $\mathtt{e}$ in formula φ. A *FOL$_=$ specification* $(\mathbb{S}, \Sigma, \Pi, \mathcal{F})$ is a FOL$_=$ signature $(\mathbb{S}, \Sigma, \Pi)$ plus a set of *closed* (i.e., no free variables) formulae $\mathcal{F}$. A *FOL$_=$ model M* is a Σ algebra together with relations for the predicate symbols in Π. Given any closed formula φ and any model M, we write $M \models \varphi$ iff M satisfies φ. If φ has free variables and $\rho : Var \rightharpoonup M$ is a partial mapping defined (at least) on all free variables in φ, also called an *M-valuation* of φ's free variables, we let $\rho \models \varphi$ denote the fact that ρ satisfies φ. Note that $\rho \models \varphi[\mathtt{e}/\mathtt{x}]$ iff $\rho[\rho(\mathtt{e})/\mathtt{x}] \models \varphi$.

IMP. Here is the syntax of our simple imperative language IMP, using BNF:

$$\begin{aligned}
Nat &::= \text{naturals},\ Nat^{+} ::= \text{positive naturals},\ Int ::= \text{integers} \\
PVar &::= \text{identifiers, to be used as } \textbf{p}\text{rogram } \textbf{var}\text{iable names} \\
E &::= Int \mid PVar \mid E_1 \;\mathtt{op}\; E_2 \\
S &::= \mathtt{skip} \mid PVar \mathrel{\mathtt{:=}} E \mid S_1 \mathtt{;} S_2 \mid \mathtt{if}\,(E)\, S_1 \;\mathtt{else}\; S_2 \mid \mathtt{while}\,(E)\, S
\end{aligned}$$

Like C, IMP has no boolean expressions (0 means "false" and $\neq 0$ means "true").

$$\boxed{\begin{aligned}
Cfg &::= \langle \mathsf{Set}[CfgItem] \rangle \\
CfgItem &::= \langle E \mid S \rangle_k \mid \langle Env \rangle_{env} \\
Env &::= \mathsf{Map}[PVar, Int]
\end{aligned}}$$

To give a matching logic semantics to a language, one needs a rigorous definition of program configurations, like in operational semantics. Besides a syntactic term, a configuration may typically contain pieces of semantic data that can be defined as conventional algebraic data types, such as associative lists (e.g., for stacks, queues), sets (e.g., for resources held), maps (e.g., for environments, stores), etc. There is no standard notation for configurations. We use the generic notation from the K framework and rewrite logic semantics [20, 28], where configurations are defined as potentially nested *cell* terms $\langle contents \rangle_{label}$ (the *label* may be omitted). The cell *contents* can be any data-structure, including, e.g., sets of other cells. Here IMP's configurations are cells of the form $\langle \langle \ldots \rangle_k \, \langle \ldots \rangle_{env} \rangle$ over the syntax in the box to the right.

How to define such cell configurations as algebraic data-types and then use them to give operational semantics to languages is explained in detail in [20, 28]. The reader may have noticed that our formal notion of configuration above is more "generous" than one may want it to be. Indeed, the syntax above allows configurations with arbitrarily many $\langle\ldots\rangle_k$ and $\langle\ldots\rangle_{env}$ cells. While multiple $\langle\ldots\rangle_k$ cells may make sense in the context of concurrent languages (where each cell $\langle\ldots\rangle_k$ would represent a concurrent thread or process; see [20, 28]), having multiple environment cells at the same level with multiple computation cells may indeed be confusing. If one wants to disallow such undesirable configurations syntactically, then one can change the grammar above accordingly; for example, one can replace the first two productions with "$C ::= \langle\langle E \mid S\rangle_k\ \langle Env\rangle_{env}\rangle$", etc.

However, as argued in [28], allowing configurations with arbitrarily many and untyped nested cells has several advantages that are hard to ignore, particularly with regards to the modularity and extendability of language definitions, so we (admittedly subjectively) prefer them in general. To ensure that bad configurations are never produced, one can check that the initial configuration is well-formed and that none of the subsequent rewrite rules changes the structure of the configuration. As an analogy, language designers often prefer to define language syntax grammars which are rather permissive, and then prove that all the semantic reduction/rewrite rules preserve a carefully chosen notion of well-definedness of the original program; this approach of not using the language syntax grammar to define well-definedness allows more flexibility in designing various type systems or abstractions over the language syntax. Configuration well-definedness "preservation" proofs would be rather easy, but we do not address them in this paper.

We do not give IMP an operational semantics here, despite being trivial, since we give it a Hoare logic semantics shortly, which suffices to later prove the soundness of our matching logic semantics as well as its relationship to Hoare logic (Thm. 1). In our companion report [27] we provide an operational semantics, together with direct proofs of soundness for the Hoare logic (Fig. 1) and matching logic (Fig. 2).

Hoare Logic Proof System for IMP. Fig. 1 gives IMP an axiomatic semantics as a Hoare logic proof system deriving (partial) correctness triples of the form $\{\varphi\}\ \mathtt{s}\ \{\varphi'\}$, where φ and φ' are FOL formulae called the *pre-condition* and the *post-condition* respectively, and $\mathtt{s}$ is a statement; the intuition for $\{\varphi\}\ \mathtt{s}\ \{\varphi'\}$ is that if φ holds before $\mathtt{s}$ is executed, then φ' holds whenever $\mathtt{s}$ terminates. In addition to the specific rules for IMP, there are generic Hoare logic rules, such as consequence or framing, that can be considered part of all Hoare logic proof systems.

$$\frac{\cdot}{\{\varphi[\mathtt{e}/\mathtt{x}]\}\ \mathtt{x}\ \mathtt{:=}\ \mathtt{e}\ \{\varphi\}}\qquad(\text{HL-}\textsc{asgn})$$

$$\frac{\{\varphi_1\}\ \mathtt{s}_1\ \{\varphi_2\},\ \{\varphi_2\}\ \mathtt{s}_2\ \{\varphi_3\}}{\{\varphi_1\}\ \mathtt{s}_1\mathtt{;}\mathtt{s}_2\ \{\varphi_3\}}\qquad(\text{HL-}\textsc{seq})$$

$$\frac{\begin{array}{c}\{\varphi \wedge (\mathtt{e} \neq 0)\}\ \mathtt{s}_1\ \{\varphi'\},\\ \{\varphi \wedge (\mathtt{e} = 0)\}\ \mathtt{s}_2\ \{\varphi'\}\end{array}}{\{\varphi\}\ \mathtt{if}\,(\mathtt{e})\,\mathtt{s}_1\,\mathtt{else}\,\mathtt{s}_2\ \{\varphi'\}}\qquad(\text{HL-}\textsc{if})$$

$$\frac{\{\varphi \wedge (\mathtt{e} \neq 0)\}\ \mathtt{s}\ \{\varphi\}}{\{\varphi\}\ \mathtt{while}\,(\mathtt{e})\ \mathtt{s}\ \{\varphi \wedge (\mathtt{e} = 0)\}}\qquad(\text{HL-}\textsc{while})$$

Fig. 1. Hoare logic formal system for IMP.

In Hoare logic, program states are mappings from variables to values, and are specified as FOL formulae—environment ρ "satisfies" φ iff $\rho \models \varphi$ in FOL. Because of this, program variables are regarded as logical variables in specifications; moreover, because of the rule (HL-ASGN) which may infuse program expression $\mathtt{e}$ into the pre-condition, specifications in Hoare logic actually extend program expressions.

The signature of the underlying FOL is the subsignature of IMP including only the expressions E and their subsorts. Its universe of variables, *Var*, is *PVar*. Assume some background theory that can prove $\mathtt{i}_1\,\mathtt{op}\,\mathtt{i}_2 = \mathtt{i}_1\ op_{Int}\ \mathtt{i}_2$ for any $\mathtt{i}_1, \mathtt{i}_2 \in Int$, where op_{Int}

is the mathematical variant of the syntactic `op`; e.g., $7 +_{Int} 2 = 9$. Expressions `e` are allowed as formulae as syntactic sugar for $\neg(\texttt{e} = 0)$. Here are two examples of correctness triples:

$$\{\texttt{x} > 0 \land \texttt{z} = \texttt{old\_z}\}\ \texttt{z := x + z}\ \{\texttt{z} > \texttt{old\_z}\}$$
$$\{\exists \texttt{z}.(\texttt{x} = 2 * \texttt{z} + 1)\}\ \texttt{x := x * x + x}\ \{\exists \texttt{z}.(\texttt{x} = 2 * \texttt{z})\}$$

The first sequent says that the new value of `z` is larger than the old value after the assignment, and the second says that if `x` is odd before the assignment then it is even after.

3 Matching Logic

We now describe matching logic and use it to give an axiomatic semantics to IMP.

Notions and Notations. To define a matching logic proof system, one needs to start with a definition of configurations. Section 2 discussed IMP's configuration as an example of a two-subcell configuration, but one can have arbitrarily complex configurations. Let $\mathcal{L} = (\Sigma_{\mathcal{L}}, \mathcal{F}_{\mathcal{L}})$ be a FOL$_=$ specification defining the configurations of some language. $\Sigma_{\mathcal{L}}$ is the signature of the language together with desired syntax for the semantic domains and data-structures, such as numbers and operations on them, lists, sets, maps, etc., as in Sec. 2, while the formulae in $\mathcal{F}_{\mathcal{L}}$ capture the desired properties of the various semantic components in the configurations, such as those of the needed mathematical domains. We may call $\mathcal{L} = (\Sigma_{\mathcal{L}}, \mathcal{F}_{\mathcal{L}})$ the *background theory*, because the formulae in $\mathcal{F}_{\mathcal{L}}$ are by default assumed to hold in any context. Let *Cfg* be the sort of configurations in $\mathcal{L}$.

Let us fix a model $\mathcal{T}_{\mathcal{L}}$ of $\mathcal{L}$, written more compactly $\mathcal{T}$. We think of the elements of $\mathcal{T}$ as concrete data, and in particular of the elements of sort *Cfg* as concrete configurations. Even though it is convenient to think of configurations and data as ground $\Sigma_{\mathcal{L}}$-terms, in this paper we impose no initiality constrains on configurations or on their constituent components; in other words, the formulae in $\mathcal{F}_{\mathcal{L}}$ are all the FOL$_=$ properties that we rely on in subsequent proofs. If one needs properties of configuration components that can only be proved by induction (e.g., commutativity of addition, associativity of list append, etc.), then one is expected to add those properties as part of $\mathcal{F}_{\mathcal{L}}$. In fact, we can theoretically assume that $\mathcal{F}_{\mathcal{L}}$ is not finite, not even enumerable. In our automated matching logic prover (see Sec. 6), $\mathcal{F}_{\mathcal{L}}$ consists of a finite set of equations, namely those that turned out to be useful in the experiments that we've done so far (our current $\mathcal{F}_{\mathcal{L}}$ is open to change and improvement); therefore, in our case we can conveniently pick $\mathcal{T}$ to be the initial model of $\mathcal{L}$, but that is not necessary. Let *Var* be a sortwise infinite set of logical, or semantical variables, and let "$\square$" be a fresh ($\square \notin \mathit{Var}$) variable of sort *Cfg*.

While in provers it is useful to make a distinction between program variables and logical variables, there is no theoretical distinction between these in Hoare logic. By contrast, in matching logic these are completely different mathematical objects: the former are syntactic constants in $\Sigma_{\mathcal{L}}$, while the latter are logical variables in *Var*.

Definition 1. *(Simple) matching logic specifications for $\mathcal{L}$, called **configuration patterns** or more directly just **patterns**, are particular FOL$_=$ formulae over the configuration signature $\Sigma_{\mathcal{L}}$ above of the form $\exists X.((\square = c) \land \varphi)$, where:*

- *$X \subset \mathit{Var}$ is the set of **(pattern) bound variables**; the remaining variables either in c or free in φ are the **(pattern) free variables**; "$\square$" appears exactly once in a pattern;*

- *We call c the **pattern structure**; it is a term of sort Cfg that may contain logical variables in Var (bound or not); it may (and typically will) also contain program variables in PVar (e.g., in an environment cell), but they are constants, not variables;*
- *φ is the **(pattern) constraint**, an arbitrary FOL$_=$ formula.*

We let Γ, Γ', ..., range over patterns. For example, the IMP pattern

$$\exists x, y.((\square = \langle\langle \texttt{x := y/x}\rangle_k \, \langle \texttt{x} \mapsto x, \texttt{y} \mapsto y, \rho\rangle_{env}\rangle) \wedge x \neq 0 \wedge y = x *_{Int} z)$$

specifies configurations whose code is "`x := y/x`" and the value held by `x` is not 0 and is precisely z times smaller than the value held by `y`; variables ρ and z are free, so they are expected to be bound by the proof context (like in Hoare logic, free variables in specifications act as proof parameters).

Note that we optionally called our matching logic specifications in Def. 1 "simple". The reason for doing so is because one can combine such specifications into more complex ones. For example, in our implementation briefly discussed in Sec. 6 we also allow *disjunctive patterns $\Gamma \vee \Gamma'$*, which are useful for case analysis. In this paper we limit ourselves to simple matching logic specifications and we drop the adjective "simple".

Let $Var^{\square}$ be the set $Var \cup \{\square\}$ of variables Var extended with the special variable $\square$ of sort Cfg. Valuations $Var^{\square} \to \mathcal{T}$ then consist of a concrete configuration γ corresponding to $\square$ and of a map $\tau : Var \to \mathcal{T}$; we write such valuations using a pairing notation, namely $(\gamma, \tau) : Var^{\square} \to \mathcal{T}$. We are now ready to introduce the concept of *pattern matching* that inspired the name of our axiomatic semantic approach.

Definition 2. *Configuration γ **matches** pattern $\exists X.((\square = c) \wedge \varphi)$ iff there is a $\tau : Var \to \mathcal{T}$ such that $(\gamma, \tau) \models \exists X.((\square = c) \wedge \varphi)$; if τ is relevant, we say that γ **τ-matches** the pattern.*

$(\gamma, \tau) \models \exists X.((\square = c) \wedge \varphi)$ is equivalent to saying that there exists $\theta_\tau : Var \to \mathcal{T}$ with $\theta_\tau\restriction_{Var\setminus X} = \tau\restriction_{Var\setminus X}$ such that $\gamma = \theta_\tau(c)$ and $\theta_\tau \models \varphi$.

We next introduce an important syntactic sugar notation for the common case when configurations are bags of cells, that is, when "$Cfg ::= \langle \mathsf{Bag}[CfgItem]\rangle$". If that is the case, then we let C, C', ..., range over configuration item bag terms, possibly with variables.

Notation 1 *If $\mathcal{L}$'s configurations have the form "$Cfg ::= \langle \mathsf{Bag}[CfgItem]\rangle$", if $\langle C\rangle$ is a Cfg-term (with variables), and if the variables in X are clear from context (say, their name starts with a "?", e.g., ?x), then we may write $\langle C \, \langle\varphi\rangle_{form}\rangle$ instead of $\exists X.((\square = \langle C\rangle) \wedge \varphi)$.*

The rationale for this notation is twofold: (1) it eases the writing and reading of matching logic proof systems by allowing meta-variables C, C' to also range over the additional subcell when not important in a given context; and (2) it prepares the ground for deriving matching logic provers from operational semantics over such cell configurations.

With this notation, the IMP pattern below Def. 1 can be written more uniformly as

$$\langle\langle \texttt{x := y/x}\rangle_k \, \langle \texttt{x} \mapsto ?x, \texttt{y} \mapsto ?y, \rho\rangle_{env} \, \langle ?x \neq 0 \wedge ?y = ?x *_{Int} z\rangle_{form}\rangle$$

Since patterns are special FOL$_=$ formulae, one can use FOL$_=$ reasoning to prove properties about patterns; e.g., one can show that the above pattern implies the one below:

$$\langle\langle \texttt{x := y/x}\rangle_k \, \langle \texttt{x} \mapsto ?x, \texttt{y} \mapsto ?x *_{Int} z, \rho\rangle_{env} \, \langle ?x \neq 0\rangle_{form}\rangle$$

A matching logic axiomatic semantics is given as a proof system for deriving sequents called *correctness pairs*, which can be thought of as "symbolic big-step sequents" as they relate configurations before the execution of a program fragment to configurations after:

Definition 3. *A matching logic* ***(partial) correctness pair*** *consists of two patterns Γ and Γ', written $\Gamma \Downarrow \Gamma'$. We call Γ a* ***pre-condition pattern*** *and Γ' a post-condition pattern.*

Assuming a hypothetical big-step operational semantics of the language under consideration, (partial) correctness pairs specify properties of big-step sequents (over concrete configurations) $\gamma \Downarrow \gamma'$, of the form "if γ τ-matches Γ then γ' τ-matches Γ'". For example, if Γ is the IMP pattern above (any of them), then $\Gamma \Downarrow \langle\langle\cdot\rangle_k\ \langle \mathtt{x} \mapsto z, ?\rho\rangle_{env}\ \langle true\rangle_{form}\rangle$ will be derivable with our matching logic proof system of IMP discussed in the sequel.

$$\frac{\models \Gamma \Rightarrow \Gamma_1,\ \ \Gamma_1 \Downarrow \Gamma_1',\ \ \models \Gamma_1' \Rightarrow \Gamma'}{\Gamma \Downarrow \Gamma'}\ \text{(ML-CONSEQ)}$$

$$\frac{\Gamma_1 \Downarrow \Gamma_2}{\xi(\Gamma_1) \Downarrow \xi(\Gamma_2)}\ \text{(ML-SUBST)}$$

$$\frac{\langle C\ \langle\mu\rangle_L\rangle \Downarrow \langle C'\ \langle\mu'\rangle_L\rangle}{\langle C\ \langle\mu, F\rangle_L\rangle \Downarrow \langle C'\ \langle\mu', F\rangle_L\rangle}\ \text{(ML-FRAME)}$$

Like in Hoare logic, most matching logic proof rules are language-specific, but there are also some general purpose proof rules. For example, the rules (ML-CONSEQ) and (ML-SUBST) in the right box (ξ is a substitution over free variables) can be used in any matching logic proof context. Also like in Hoare logic, the formulae in the background theory can be used when proving implications. Theoretically, this is nothing special because one can always assume that each pattern constraint also includes the background formulae $\mathcal{F}_{\mathcal{L}}$; practically, one would probably prefer to minimize the complexity of the patterns and thus keep the background theory separate, making use of its formulae whenever needed. Unlike in Hoare logic, in matching logic one can also add framing rules for any of the cells in one's language configurations, not only for the $\langle\ldots\rangle_{form}$ cell. The rule (ML-FRAME) in the box above-right shows a generic framing rule for a cell L (for notational simplicity, we used the cell-based Notation 1; if L is *form*, then _, _ is _ ∧ _). One should add framing rules on a case-by-case basis; for some complex configurations, one may not want to add framing rules for all cells. More details about framing and other generic rules can be found in our technical report [26].

Matching Logic Proof System for IMP. Figure 2 gives a matching logic proof system for IMP. To make it resemble the more familiar Hoare logic system (Fig. 1), we adopt the following notations:

$$\begin{array}{ll}
(C\ \langle\rho\rangle_{env})[\mathtt{x} \leftarrow v] & \text{for } C\ \langle\rho[v/\mathtt{x}]\rangle_{env} \\
(C\ \langle\varphi\rangle_{form}) \wedge \varphi' & \text{for } C\ \langle\varphi \wedge \varphi'\rangle_{form} \\
C[\mathtt{e}] \equiv v & \text{for } \langle\langle\mathtt{e}\rangle_k\ C\rangle \Downarrow \langle\langle v\rangle_k\ C\rangle \\
\langle C\rangle\ \mathtt{s}\ \langle C'\rangle & \text{for } \langle\langle\mathtt{s}\rangle_k\ C\rangle \Downarrow \langle\langle\cdot\rangle_k\ C'\rangle
\end{array}$$

The meta-variables C, C', C_1, C_2 above and in Fig. 2 range over appropriate configuration item bag terms so that the resulting patterns are well-formed.The first notation can be included as an operation in the algebraic signature or IMP's configurations; the third works because IMP's expressions are side-effect-free (so C does not change). In the case of IMP, configurations have only two subcells, the code and an environment. Using generic meta-variables like C instead of more concrete configuration item terms is key to the modularity of matching logic definitions and proofs. Indeed, to add heaps to IMP

$$\frac{C[\mathtt{e}] \equiv v}{\langle C\rangle\ \mathtt{x := e}\ \langle C[\mathtt{x} \leftarrow v]\rangle} \quad \text{(ML-ASGN)}$$

$$\frac{\langle C_1\rangle\ \mathtt{s}_1\ \langle C_2\rangle,\quad \langle C_2\rangle\ \mathtt{s}_2\ \langle C_3\rangle}{\langle C_1\rangle\ \mathtt{s}_1\mathtt{;}\mathtt{s}_2\ \langle C_3\rangle} \quad \text{(ML-SEQ)}$$

$$\frac{C[\mathtt{e}] \equiv v,\quad \langle C \wedge (v \neq 0)\rangle\ \mathtt{s}_1\ \langle C'\rangle,\quad \langle C \wedge (v = 0)\rangle\ \mathtt{s}_2\ \langle C'\rangle}{\langle C\rangle\ \mathtt{if\,(e)}\ \mathtt{s}_1\ \mathtt{else}\ \mathtt{s}_2\ \langle C'\rangle} \quad \text{(ML-IF)}$$

$$\frac{C[\mathtt{e}] \equiv v,\quad \langle C \wedge (v \neq 0)\rangle\ \mathtt{s}\ \langle C\rangle}{\langle C\rangle\ \mathtt{while\,(e)\ s}\ \langle C \wedge (v = 0)\rangle} \quad \text{(ML-WHILE)}$$

$$\frac{\cdot}{C[\mathtt{i}] \equiv \mathtt{i}} \quad \text{(ML-INT)}$$

$$\frac{\cdot}{(C\ \langle \mathtt{x} \mapsto v, \rho\rangle_{env})[\mathtt{x}] \equiv v} \quad \text{(ML-LOOKUP)}$$

$$\frac{C[\mathtt{e}_1] \equiv v_1,\quad C[\mathtt{e}_2] \equiv v_2}{C[\mathtt{e}_1\ \mathtt{op}\ \mathtt{e}_2] \equiv v_1\ op_{Int}\ v_2} \quad \text{(ML-OP)}$$

Fig. 2. IMP matching logic formal system

in Sec. 5 we only add new rules for the new language constructs (none of the rules in Fig. 2 changes), and let C include an additional heap cell.

The rule (ML-LOOKUP) in Fig. 2, which desugared says $\langle\langle \mathtt{x}\rangle_k\ \langle \mathtt{x} \mapsto v, \rho\rangle_{env}\ C\rangle_{config} \Downarrow \langle\langle v\rangle_k\ \langle \mathtt{x} \mapsto v, \rho\rangle_{env}\ C\rangle_{config}$ is derivable, shows an interesting and common situation where a configuration contains a term which, in order to make sense, needs to satisfy additional constraints. Indeed, the term "$\mathtt{x} \mapsto v, \rho$" appears in the $\langle \ldots\rangle_{env}$ cell which wraps a map structure, so one expects that $\mathtt{x}$ is not in the domain of ρ. To avoid notational clutter, we always assume that sufficient conditions are given in configurations' constraints so that each term appearing in any configuration is well-defined, according to its sort's corresponding notion of well-definedness in the background theory, *if any*. If the background theory does not axiomatize well-definedness for certain sorts, then one may miss the opportunity to derive certain correctness pairs, but it is important to understand that one cannot derive wrong facts. Indeed, the correctness pair above is sound no matter whether C's constraint states that $\mathtt{x}$ is not in the domain of ρ or not, because if $\mathtt{x}$ is in the domain of ρ then no concrete configuration will ever match the pre-condition.

Despite its operational flavor (e.g., see (ML-ASGN)), the matching logic proof system is as rigorous and compositional as the Hoare logic one; in particular, the rule (ML-WHILE) is almost identical to (HL-WHILE). The soundness of this particular matching logic proof system follows directly from Thm. 1 and the soundness of the Hoare logic proof system in Fig. 1. We additionally have a direct proof of soundness, with respect to an operational semantics, that can be found in our technical report [27].

4 Equivalence of Hoare and (a Restricted Form of) Matching Logic

We show that, for IMP, any property provable using Hoare logic is also provable using matching logic and vice-versa. Our proof reductions are mechanical in both directions, which means that one can automatically generate a matching logic proof from any Hoare logic proof and vice-versa. For the embedding of Hoare logic into matching logic part we do not use the fact that the configuration contains only an environment and a computation, so this result also works for other languages that admit Hoare logic proof systems.

Before we proceed with the technical constructions, we need to impose a restriction on the structure of matching logic patterns to be used throughout this section. Note that a pattern of the form $\langle C\, \langle \mathtt{x} \mapsto x\rangle_{env}\rangle$ specifies configurations whose environments *only* declare x (and its value is x), while a pattern $\langle C\, \langle\cdot\rangle_{env}\rangle$ specifies configurations with empty (or "$\cdot$") environments. Thus, one is able to derive $\langle\langle \mathtt{x} \mapsto x\rangle_{env}\rangle$ $\mathtt{x := x - x}$ $\langle\langle \mathtt{x} \mapsto 0\rangle_{env}\rangle$, but it is impossible to derive $\langle\langle\cdot\rangle_{env}\rangle$ $\mathtt{x := x - x}$ $\langle\langle \mathtt{x} \mapsto 0\rangle_{env}\rangle$: one will never be able to "evaluate" x in the empty environment. However, note that the obvious Hoare logic equivalent, namely $\{true\}$ $\mathtt{x := x - x}$ $\{\mathtt{x} = 0\}$, is unconditionally derivable. To avoid such situations, we: (1) fix a finite set of program variables $\mathtt{Z} \subset PVar$ which is large enough to include all the program variables that appear in the original program that one wants to verify; and (2), restrict the IMP matching logic patterns to ones whose environments have the domain precisely Z. The need for this restriction in order to prove the equivalence of the two formal systems suggests that matching logic patterns allow for more informative specifications than Hoare logic. Also, we assume that $Z \subseteq Var$ is a set of "semantic clones" of the program variables in Z, that is, $Z = \{z \mid \mathtt{z} \in \mathtt{Z}\}$, and that the semantic variables in Z are reserved only for this purpose. Also, let ρ_Z be the special environment mapping each program variable $\mathtt{z} \in \mathtt{Z}$ into its corresponding semantic clone $z \in Z$.

We first define mappings *H2M* and *M2H* taking Hoare logic correctness triples to matching logic correctness pairs, and matching logic correctness pairs to Hoare logic correctness triples, respectively. Then we show in Thm. 1 that these mappings are logically inverse to each other and that they take derivable sequents in one logic to derivable sequents in the other logic; for example, if a correctness triple $\{\varphi\}$ s $\{\varphi'\}$ is derivable with the Hoare logic proof system in Fig. 1, then the correctness pair $H2M(\{\varphi\}\ \mathtt{s}\ \{\varphi'\})$ is derivable with the matching logic proof system in Fig. 2.

4.1 *H2M*: From Hoare to Matching Logic

Hoare logic makes no distinction between program and logic variables. Let variables in Hoare specifications but not in the original program be semantic variables in *Var*. Let $H2M(\varphi, \mathtt{s})$ be an auxiliary map taking formulae φ and statements s to patterns as follows:

$$H2M(\varphi, \mathtt{s}) \overset{\text{def}}{=} \exists Z.((\square = \langle\langle \mathtt{s}\rangle_k\, \langle \rho_Z\rangle_{env}\rangle) \wedge \rho_Z(\varphi))$$

Hence, $H2M(\varphi, \mathtt{s})$ is a pattern whose code is s, whose environment ρ_Z maps each $\mathtt{z} \in \mathtt{Z}$ in φ or in s into its semantic clone $z \in Z$, and whose constraint $\rho_Z(\varphi)$ renames all the program variables in φ into their semantic counterparts. We now define the mapping from Hoare logic correctness triples into matching logic correctness pairs as follows:

$$H2M(\{\varphi\}\ \mathtt{s}\ \{\varphi'\}) \overset{\text{def}}{=} H2M(\varphi, \mathtt{s}) \Downarrow H2M(\varphi', \cdot)$$

For example, if $\mathtt{Z} = \{\mathtt{x}, \mathtt{z}\}$ then $H2M(\{\mathtt{x} > 0 \wedge \mathtt{z} = u\}\ \mathtt{z := x + z}\ \{\mathtt{z} > u\})$ is

$$\exists x, z.((\square = \langle\langle \mathtt{z := x + z}\rangle_k\, \langle \mathtt{x} \mapsto x,\ \mathtt{z} \mapsto z\rangle_{env}\rangle) \wedge x > 0 \wedge z = u)$$
$$\Downarrow \exists x, z.((\square = \langle\langle \mathtt{skip}\rangle_k\, \langle \mathtt{x} \mapsto x,\ \mathtt{z} \mapsto z\rangle_{env}\rangle) \wedge z > u)$$

The resulting correctness pairs are quite intuitive, making use of pattern bound variables as a bridge between the program variables and the semantic constraints on them.

4.2 *M2H*: From Matching to Hoare Logic

Given an environment $\rho = (\mathtt{x_1} \mapsto v_1,\ \mathtt{x_2} \mapsto v_2,\ \ldots,\ \mathtt{x_n} \mapsto v_n)$, let $\overline{\rho}$ be the FOL$_=$ formula $\mathtt{x_1} = v_1 \wedge \mathtt{x_2} = v_2 \wedge \ldots \wedge \mathtt{x_n} = v_n$. We define the mapping *M2H* taking matching logic statement correctness pairs into Hoare logic correctness triples as follows:

$$M2H(\exists X.((\square = \langle\langle \mathtt{s}\rangle_k\, \langle\rho\rangle_{env}\rangle) \wedge \varphi) \Downarrow \exists X'.((\square = \langle\langle \mathtt{skip}\rangle_k\, \langle\rho'\rangle_{env}\rangle) \wedge \varphi'))$$
$$= \{\exists X.(\overline{\rho} \wedge \varphi)\}\ \mathtt{s}\ \{\exists X'.(\overline{\rho'} \wedge \varphi')\}$$

For example, if $\Gamma \Downarrow \Gamma'$ is the correctness pair in Sec. 4.1, then $M2H(\Gamma \Downarrow \Gamma')$ is

$$\{\exists x, z.(\mathtt{x} = x \wedge \mathtt{z} = z \wedge x > 0 \wedge z = u)\}\ \mathtt{z := x + z}\ \{\exists x, z.(\mathtt{x} = x \wedge \mathtt{z} = z \wedge z > u)\}$$

We say that two FOL formulae φ_1 and φ_2 are logically equivalent iff $\models \varphi_1 \Leftrightarrow \varphi_2$. Moreover, correctness triples $\{\varphi_1\}\ \mathtt{s}\ \{\varphi'_1\}$ and $\{\varphi_2\}\ \mathtt{s}\ \{\varphi'_2\}$ are logically equivalent iff $\models \varphi_1 \Leftrightarrow \varphi_2$ and $\models \varphi'_1 \Leftrightarrow \varphi'_2$; similarly, matching logic correctness pairs $\Gamma_1 \Downarrow \Gamma'_1$ and $\Gamma_2 \Downarrow \Gamma'_2$ are logically equivalent iff $\models \Gamma_1 \Leftrightarrow \Gamma_2$ and $\models \Gamma'_1 \Leftrightarrow \Gamma'_2$. Thanks to the rules (HL-conseq) and (ML-conseq), logically equivalent sequents are either both or none derivable. Since $\exists x, z.(\mathtt{x} = x \wedge \mathtt{z} = z \wedge x > 0 \wedge z = u)$ is logically equivalent to $\mathtt{x} > 0 \wedge \mathtt{z} = u$ and since $\exists z.(\mathtt{z} = z \wedge z > u)$ is logically equivalent to $\mathtt{z} > u$, we can conclude that the correctness triple $M2H(\Gamma \Downarrow \Gamma')$ above is logically equivalent to $\{\mathtt{x} > 0 \wedge \mathtt{z} = u\}\ \mathtt{z := x + z}\ \{\mathtt{z} > u\}$.

Theorem 1 (Equivalence of Matching Logic and Hoare Logic for IMP). *Any given Hoare triple $\{\varphi\}\ \mathtt{s}\ \{\varphi'\}$ is logically equivalent to $M2H(H2M(\{\varphi\}\ \mathtt{s}\ \{\varphi'\}))$, and any matching logic correctness pair $\Gamma \Downarrow \Gamma'$ is logically equivalent to $H2M(M2H(\Gamma \Downarrow \Gamma'))$. Moreover, for any Hoare logic proof of $\{\varphi\}\ \mathtt{s}\ \{\varphi'\}$ one can construct a matching logic proof of $H2M(\{\varphi\}\ \mathtt{s}\ \{\varphi'\})$, and for any matching logic proof of $\Gamma \Downarrow \Gamma'$ one can construct a Hoare logic proof of $M2H(\Gamma \Downarrow \Gamma')$.* *(Proof given in [27])*

5 Adding a Heap

We next work with HIMP (IMP with a heap), an extension of IMP with dynamic memory allocation/deallocation and pointer arithmetic. We show that the IMP matching logic formal system extends modularly to HIMP. The heap allows for introducing and axiomatizing heap data-structures by means of pointers, such as lists, trees, graphs, etc. Unlike in separation logic where such data-structures are defined by means of recursive predicates, we define them as ordinary term constructs, with natural FOL$_=$ axioms saying how they can be identified and/or manipulated in configurations or patterns.

$$\frac{(\langle\rho\rangle_{env}\ \langle\sigma\rangle_{mem}\ C)[\overline{\mathtt{e}}] \equiv \overline{v},\ \text{ where } ?p \in \mathit{Var} \text{ is a fresh variable}}{\langle\langle\rho\rangle_{env}\ \langle\sigma\rangle_{mem}\ C\rangle\ \mathtt{x} := \mathtt{cons}(\overline{\mathtt{e}})\ \langle\langle\rho[?p/\mathtt{x}]\rangle_{env}\ \langle ?p \mapsto [\overline{v}],\ \sigma\rangle_{mem}\ C \wedge (?p \geq 0)\rangle} \quad (\text{ML-}\textsc{cons})$$

$$\frac{(\langle v \mapsto v',\ \sigma\rangle_{mem}\ C)[\mathtt{e}] \equiv v}{\langle\langle v \mapsto v',\ \sigma\rangle_{mem}\ C\rangle\ \mathtt{dispose(e)}\ \langle\langle\sigma\rangle_{mem}\ C\rangle} \quad (\text{ML-}\textsc{dispose})$$

$$\frac{(\langle\rho\rangle_{env}\ \langle v \mapsto v',\ \sigma\rangle_{mem}\ C)[\mathtt{e}] \equiv v}{\langle\langle\rho\rangle_{env}\ \langle v \mapsto v',\ \sigma\rangle_{mem}\ C\rangle\ \mathtt{x} := [\mathtt{e}]\ \langle\langle\rho[v'/\mathtt{x}]\rangle_{env}\ \langle v \mapsto v',\ \sigma\rangle_{mem}\ C\rangle} \quad (\text{ML-}[\textsc{lookup}])$$

$$\frac{(\langle v_1 \mapsto v_2',\ \sigma\rangle_{mem}\ C)[(\mathtt{e}_1, \mathtt{e}_2)] \equiv (v_1, v_2)}{\langle\langle v_1 \mapsto v_2',\ \sigma\rangle_{mem}\ C\rangle\ [\mathtt{e}_1] := \mathtt{e}_2\ \langle\langle v_1 \mapsto v_2,\ \sigma\rangle_{mem}\ C\rangle} \quad (\text{ML-}[\textsc{mutate}])$$

Fig. 3. HIMP Matching logic formal system (these rules to be added to those in Fig. 2)

$$\begin{array}{l}
S \ ::=\ \ldots \mid \mathit{PVar} := \mathsf{cons}(\mathsf{List}[E]) \\
\quad \mid \mathtt{dispose}(E) \\
\quad \mid \mathit{PVar} := [E] \\
\quad \mid [E_1] := E_2 \\
\mathit{CfgItem} ::=\ \ldots \mid \langle \mathit{Mem}\rangle_{mem} \\
\mathit{Mem} ::=\ \mathsf{Map}[\mathit{Nat}^+, \mathit{Int}]
\end{array}$$

HIMP Configuration. The configuration of HIMP extends that of IMP with a heap, or memory cell, as shown in the right box. A heap is a (partial) map structure just like the environment, but from positive naturals (also called pointers) to integers. We use no special connective to construct heaps, as they are simply maps like any other. Our heap-related constructs are based on those described by Reynolds [25]. The `cons` construct is used to simultaneously allocate memory and assign to it, while $[E]$ is used for lookup when on the right-hand side of an assignment, and mutation when on the left-hand side. This is very much like the $*$ operator in C. Finally, $\mathtt{dispose}(E)$ removes a single mapping from the heap.

Matching Logic Definition of HIMP. Figure 3 shows the rules that need to be added to those in Fig. 2 to obtain a matching logic formal system for HIMP. Since patterns inherit the structure of configurations, matching logic is as modular as the underlying configuration. In particular, none of the matching logic rules in Fig. 2 need to change. To obtain a matching logic semantics for HIMP, all we need to add is one rule for each new language construct, as shown in Fig. 3. To save space, we write "$C[(\mathtt{e}_1, \mathtt{e}_2, \ldots, \mathtt{e}_\mathtt{n})] \equiv (v_1, v_2, \ldots, v_n)$" for "$C[\mathtt{e}_1] \equiv v_1$ and $C[\mathtt{e}_2] \equiv v_2$ and ... and $C[\mathtt{e}_\mathtt{n}] \equiv v_n$" and "$?p \mapsto [v_1, ..., v_n]$" instead of "$?p \mapsto v_1, ..., ?p \mapsto v_n$". One can now easily derive, e.g.,

$$\begin{array}{l}
\langle\langle \mathtt{x} := \mathtt{cons}(1, \mathtt{x}); [\mathtt{x}] := [\mathtt{x} + 1]; \mathtt{dispose}(\mathtt{x} + 1)\rangle_k\ \langle \mathtt{x} \mapsto x, \rho\rangle_{env}\ \langle\sigma\rangle_{mem}\ \langle\varphi\rangle_{form}\rangle \\
\Downarrow \langle\langle\cdot\rangle_k\ \langle \mathtt{x} \mapsto ?p, \rho\rangle_{env}\ \langle ?p \mapsto x, \sigma\rangle_{mem}\ \langle\varphi \wedge ?p \geq 0\rangle_{form}\rangle
\end{array}$$

where x, ρ and σ are free variables of appropriate sorts and $?p$ is a bound variable.

As we already mentioned, to keep the presentation simple we assume that a pattern's constraint is strong enough to guarantee that each term appearing in the pattern is well-defined. That is the reason for which we added the constraint $p? \geq 0$ to the post-condition of the bottom correctness pair in (ML-CONS) in Fig. 3, and also the reason for which we did not require that the pre-conditions' constraints in the rules (ML-DISPOSE), (ML-LOOKUP) and (ML-MUTATE) guarantee that v (first two) and v' (the third), resp., are not in the domain of σ. As discussed at the end of Sec. 3, matching logic proof rules are still sound even if the patterns' constraints do not guarantee well-definedness of all the terms appearing in the pattern, but if this information is missing then one may not be able to derive certain correctness pairs that would otherwise be derivable. In our implementation, we prefer to keep the formula in $\langle\ldots\rangle_{form}$ free of well-definedness information and, instead, collect that information by need from the rest of the pattern; e.g., we do not need to add the constraint $?p \geq 0$ in our implementation of (ML-CONS).

Defining and Using Heap Patterns. Most complex programs organize heap data in structures such as linked lists, trees, graphs, etc. To verify such programs, one needs to be able to specify and reason about heap structures. Consider linked lists whose nodes consist of two consecutive locations: an integer (data) followed by a pointer to next node (or 0 for the list end). One is typically interested in reasoning about the sequences of integers held by such list structures. It is then natural to define a list heap constructor "list : $Nat \times \mathsf{List}[Int] \to Mem$" taking a pointer (the location where the list starts) and a sequence of integers (the data held by the list, with "ϵ" for the empty sequence and ":" for sequence concatenation) and yielding a fragment of memory. It does not make sense to define this as one would a function, since it is effectively non-deterministic, but it can be axiomatized as a FOL$_=$ formula as follows, in terms of patterns:

$$\begin{aligned}\langle\langle \mathrm{list}(p,\alpha),\ \sigma\rangle_{mem}\ \langle\varphi\rangle_{form}\ C\rangle \Leftrightarrow\ & \langle\langle\sigma\rangle_{mem}\ \langle p = 0 \wedge \alpha = \epsilon \wedge \varphi\rangle_{form}\ C\rangle \\ \vee\ & \langle\langle p \mapsto [?a,?q],\ \mathrm{list}(?q,?\beta),\ \sigma\rangle_{mem}\ \langle \alpha = ?a{:}?\beta \wedge \varphi\rangle_{form}\ C\rangle\end{aligned}$$

In words, a list pattern can be identified in the heap starting with pointer p and containing integer sequence α iff either the list is empty, so it takes no memory and its pointer is null (0), or the list is non-empty, so it holds its first element at location p and a pointer to a list containing the remaining elements at location $p + 1$. Using this axiom, one can prove properties about patterns, such as:

$$\begin{aligned}&\langle\langle 5 \mapsto 2,\ 6 \mapsto 0,\ 8 \mapsto 3,\ 9 \mapsto 5,\ \sigma\rangle_{mem}\ C\rangle \Rightarrow \langle\langle \mathrm{list}(8, 3:2),\ \sigma\rangle_{mem}\ C\rangle, \quad \text{and}\\ &\langle\langle \mathrm{list}(8, 3:2),\ \sigma\rangle_{mem}\ C\rangle \Rightarrow \langle\langle 8 \mapsto 3,\ 9 \mapsto ?q,\ ?q \mapsto 2,\ ?q + 1 \mapsto 0,\ \sigma\rangle_{mem}\ C\rangle\end{aligned}$$

Similar axiomatizations are given in [27] for other heap patterns (trees, queues, graphs) and are supported by our current matching logic verifier (briefly discussed below).

It is worthwhile emphasizing that we are here talking about axiomatic, as opposed to constructive, definitions of heap patterns. Axioms like the one for lists above are simply added to the background theory. Like with the other axioms in the background theory, one can attempt to prove them from basic principles of mathematics and constructive definitions if one wants to have full confidence in the results of verification.

6 Proof Example and Practical Experience

In order to give a more practical understanding of matching logic, here we describe a concrete example as well as a quick description of our verification results using a matching logic verifier for a fragment of C implemented in Maude, which is available for download and online execution at `http://fsl.cs.uiuc.edu/ml`.

List-Reverse Example. Consider proving that a program correctly reverses a list. A similar proof, but using separation logic, is given by Reynolds in [25]. Given `x` pointing to the beginning of a linked list, the following HIMP program reverses that list in-place:

```
p:=0; while (x!=0) ( y:=[x+1]; [x+1]:=p; p:=x; x:=y )
```

We assume each list node is two contiguous memory locations, the first containing the value, the second containing a pointer to the next node. Initially `[x]` is the value of the first node and `[x + 1]` points to the second node in the list.

Matching logic uses the standard notion of loop-invariants to prove loops correct. The fundamental idea is to find one pattern that is always true when leaving the loop, whether the loop condition is true or false. As before, the order of the configuration pieces does not matter. The invariant configuration for our reverse program can be:

$$\langle\langle \mathtt{p} \mapsto ?p,\ \mathtt{x} \mapsto ?x,\ \mathtt{y} \mapsto ?x\rangle_{env}\ \langle \mathrm{list}(?p, ?\beta),\ \mathrm{list}(?x, ?\gamma)\rangle_{mem}\ \langle \mathrm{rev}(\alpha) = \mathrm{rev}(?\gamma){:}?\beta\rangle_{form}\rangle$$

where the environment binds program variable `p` to pointer $?p$ and program variables `x` and `y` to the same value $?x$. In the memory we see list($?p$, $?\beta$) and list($?x$, $?\gamma$)—two disjoint lists, the first starting with pointer $?p$ and holding sequence $?\beta$ and the second starting with pointer $?x$ and holding sequence $?\gamma$. Unlike in separation logic where "list" needs to be a predicate holding in a separate portion of the heap, in matching logic "list" is an ordinary operation symbol added to the signature and constrained through axioms as shown in Sec. 5. The pattern formula guarantees that $\mathrm{rev}(\alpha) = \mathrm{rev}(?\gamma){:}?\beta$, where α, the original sequence of integers in the list pointed to by `x`, is the only non-bound variable in the pattern. Now we see how the pattern progresses as we move into the loop:

$$\langle\langle \mathtt{p} \mapsto ?p, \mathtt{x} \mapsto ?x, \mathtt{y} \mapsto ?x\rangle_{env}\ \langle \mathrm{list}(?p,?\beta),\ \mathrm{list}(?x,?\gamma)\rangle_{mem}\ \langle \mathrm{rev}(\alpha) = \mathrm{rev}(?\gamma){:}?\beta \wedge \boldsymbol{?x \neq 0}\rangle_{form}\rangle$$

Note, we use **bold** to indicate changes. Inside the body of the while loop, we know that the guarding condition is true, so we assume it by adding it to our formula. Now that we know $?x$ is not nil, we can expand the definition of the list list($?x$, $?\gamma$) in the heap and thus yield:

$$\left\langle \begin{array}{c} \langle \mathtt{p} \mapsto ?p,\ \mathtt{x} \mapsto ?x,\ \mathtt{y} \mapsto ?x\rangle_{env}\ \langle \mathrm{list}(?p, ?\beta),\ \boldsymbol{?x \mapsto [?a, ?x'],\ \mathbf{list}(?x', ?\gamma')}\rangle_{mem} \\ \langle \mathrm{rev}(\alpha) = \mathrm{rev}(?\gamma){:}?\beta \wedge ?x \neq 0\ \wedge\ \boldsymbol{?\gamma = ?a{:}?\gamma'}\rangle_{form} \end{array} \right\rangle$$

This pattern now contains all the configuration infrastructure needed to process the four assignments `y:=[x+1]; [x+1]:=p; p:=x; x:=y`, which yield the following pattern:

$$\left\langle \begin{array}{c} \boldsymbol{\langle \mathtt{p} \mapsto ?x,\ \mathtt{x} \mapsto ?x',\ \mathtt{y} \mapsto ?x'\rangle_{env}}\ \langle \mathrm{list}(?p, ?\beta),\ \boldsymbol{?x \mapsto [?a, ?p]},\ \mathbf{list}\boldsymbol{(?x', ?\gamma')}\rangle_{mem} \\ \langle \mathrm{rev}(\alpha) = \mathrm{rev}(?\gamma){:}?\beta \wedge ?x \neq 0 \wedge ?\gamma = ?a{:}?\gamma'\rangle_{form} \end{array} \right\rangle$$

The list axiom can be applied again to the resulting pattern to reshape its heap into one having a list at $?x$. We can additionally use the fact that $?\gamma = ?a{:}?\gamma'$ to rewrite our $\mathrm{rev}(\alpha) = \mathrm{rev}(?\gamma){:}?\beta$ to $\mathrm{rev}(\alpha) = \mathrm{rev}(?a{:}?\gamma'){:}?\beta$. The axioms for reverse then tell us this is equivalent to $\mathrm{rev}(\alpha) = \mathrm{rev}(?\gamma'){:}?a{:}?\beta$. We therefore obtain the following pattern:

$$\left\langle \begin{array}{c} \langle \mathtt{p} \mapsto ?x,\ \mathtt{x} \mapsto ?x',\ \mathtt{y} \mapsto ?x'\rangle_{env}\ \langle \mathbf{list}\boldsymbol{(?x, ?a{:}?\beta)},\ \mathrm{list}(?x', ?\gamma')\rangle_{mem} \\ \langle \mathbf{rev}\boldsymbol{(\alpha) = }\mathbf{rev}\boldsymbol{(?\gamma'){:}?a{:}?\beta}\ \wedge\ ?x \neq 0\rangle_{form} \end{array} \right\rangle$$

Now we are in a position to prove that this pattern logically implies the original invariant. Recall that bound variables are quantified existentially. It is then easy to see that since this is a more specific pattern than the invariant itself, the invariant follows logically from this pattern. Thus, we have shown that the invariant always holds after exiting the loop.

```
     assume
       <config>
         <env> p |-> ?p, x |-> ?x, y |-> ?y </env>
         <heap> list(?x)(A) </heap>
         <form> TrueFormula </form>
       </config> ;
p = 0 ;
     invariant
       <config>
         <env> p |-> ?p, x |-> ?x, y |-> ?y </env>
         <heap> list(?p)(?B), list(?x)(?C) </heap>
         <form> rev(A)===rev(?C)::?B </form>
       </config> ;
while(x != 0) {
  y = *(x + 1) ;
  *(x + 1) = p ;
  p = x ;
  x = y ;
}
     assert
       <config>
         <env> p |-> ?p, x |-> ?x, y |-> ?y </env>
         <heap> list(?p)(rev(A)) </heap>
         <form> TrueFormula </form>
       </config> ;
```

Fig. 4. The reverse example in our matching logic prover

Experience with Our Verifier. We have implemented a matching logic verifier for a fragment of C, using Maude [4]. Since matching logic is so close to operational semantics, our implementation essentially modifies a rewriting logic executable semantics of the language, in the style presented in [20, 28], turning it into a matching logic prover. Our prover extends the language with `assume`, `assert`, and `invariant` commands, each taking a pattern. For example, Figure 4 shows the list-reverse example above, as given to our verifier. After execution, the following result is output to the user:

```
rewrites: 3146 in 5ms cpu (5ms real) (524420 rewrites/second)
result Result: 2 feasible and 3 infeasible paths
```

Our verifier is path-based, cutting paths as quickly as found infeasible. Each assertion (including the two implicit ones associated to an invariant) results in a proof obligation, namely that the current pattern matches (or implies) the asserted one. To prove such implications, we implemented a simple $\mathrm{FOL}_=$ prover which: (1) skolemizes the variables bound in the hypothesis pattern; (2) iterates through each subcell in the hypothesis pattern and attempts to match its contents against the corresponding cell in the conclusion, this way accumulating a substitution of the variables bound in the conclusion pattern; (3) the computed substitution is applied on the fly, together with equational simplification axioms for various mathematical domains (such as integer arithmetic, lists, trees, etc.); (4) the remaining matching tasks which cannot be solved using Maude's term matching and the above substitution propagation, are added to the constraints of the conclusion pattern; (5) eventually, all is left is the two $\langle\ldots\rangle_{form}$ cells, which may generate an implication of formulae over various domains; if that is the case, we send the resulting formula to the Z3 SMT solver (we have modified/recompiled Maude for this purpose). In most of our experiments, including the one in Fig. 4, our Maude domain axiomatizations were sufficient to discard all proof obligations without a need for an external SMT solver.

One can download or use our matching logic verifier through an online interface at `http://fsl.cs.uiuc.edu/ml`. The following examples are available on the website and can be verified in less than 1 second (all of them, together; we use a 2.5GHz Linux machine to run the online interface): the sum of numbers from 1 to n, three variants of list reverse, list append, list length, queue enqueuing and dequeuing, transferring elements from one queue to another (using transfer of ownership and stealing), mirroring a tree, converting (the leaves of) a tree to a list, and insertion-, bubble-, quick- and merge-sort using linked lists as in this paper. All proofs are of complete correctness, not only memory safety. Additionally, both aspects are always verified using a single prover—there is no need to have one prover to verify correctness and another to verify memory safety. In fact, matching logic makes no distinction between the two kinds of correctnesses.

We have also verified the Schorr-Waite graph marking algorithm [13] (used in garbage collection); the details can be found in [27]. Our verifier automatically generated and proved all 227 paths in a few seconds. The formalization uses novel axiomatizations of clean and marked partial graphs, as well as a specific stack-in-graph structure, which records that during the execution of Schorr-Waite the heap consists at any time of such a stack and either a clean or a marked partial subgraph. To the best of our knowledge, this is the first time the partial correctness of this algorithm has been automatically verified. We use the word "automatically" in the sense that no user intervention is required to make the proof go through other than correctly describing the invariant. Previous automated efforts have either proved only its memory safety [16] or a version restricted to trees [19] automatically.

7 Conclusion, Related Work and Future Work

This paper introduced matching logic and showed how it relates to the most traditional logics for program verification, Hoare logic. Matching logic program specifications are constrained algebraic structures, called *patterns*, formed by allowing (and constraining) variables in program configurations. Configurations satisfy patterns iff they match their structure consistently with their constraints. Matching logic formal systems mimic (big-step) operational semantics, making them relatively easy to turn into forwards-analysis program verifiers. However, specifications tend to be more verbose in matching logic than in Hoare logic, because one may need to mention a larger portion of the program configuration. On the other hand, matching logic appears to handle language extensions better than Hoare logic, precisely because it has direct access to the program configuration.

Matching logic is related to many verification approaches; here we only briefly discuss its relationships to Floyd/Hoare logics, evolving algebra/specifications, separation logic, shape analysis, and dynamic logic. There are many Hoare-logic-based verification frameworks, such as ESC/Java [9], Spec# tool [1], HAVOC [17], and VCC [5]. Caduceus/Why [8, 16] proved many properties relating to the Schorr-Waite algorithm. However, their proofs were not entirely automated. The weakness of traditional Hoare-like approaches is that reasoning about non-inductively defined data-types and about heap structures tend to be difficult, requiring extensive manual intervention in the proof process.

The idea of regarding a program (fragment) as a specification transformer to analyze programs in a forwards-style is very old. In fact, Floyd did precisely that in his seminal 1967 paper [10]: if φ holds before `x := e` is executed, then $\exists v.\ (\mathtt{x} = \mathtt{e}[v/\mathtt{x}]) \wedge \varphi[v\,/\,\mathtt{x}]$ holds after. Thus, the assignment statement can be naturally regarded as a transition, or a rewrite, from one FOL formula to another. Equational algebraic specifications have also been used to express pre- and post-conditions and then verify programs in a forwards manner using term rewriting [11]. Evolving specifications [24], building upon intuitions from evolving algebra and ASMs, adapt and extend this basic idea to compositional systems, refinement and behavioral specifications. Many other verification approaches, some discussed above or below, can be cast as formula-transforming ones, and matching logic makes no exception. What distinguishes the various approaches is the formalism and style used for specifications. What distinguishes matching logic is its apparently very low level formalism, which drops no detail from the program configuration, which makes it resemble operational semantics. The other approaches we are aware of attempt to do the opposite, namely to use formalisms which are as abstract as possible. Matching logic builds upon the belief that there are some advantages of working with explicit configuration patterns instead of abstract formulae, and that the use of symbolic variables in configurations can still offer a comfortable level of abstraction by only mentioning in each rule those configuration components which are necessary.

Separation logic [23, 25] is an extension of Hoare logic. There are many variants and extensions of separation logic that we cannot discuss here. There is a major difference between separation and matching logic: the former extends Hoare logic to work better with heaps, while matching logic attempts to provide an alternative to Hoare logics in which the program configuration structure is explicit in the specifications, so heaps are treated uniformly just like any other structures in the configuration. Smallfoot [3] and jStar [7] are separation logic tools with good support for proving memory safety.

Shape analysis [29] allows one to examine and verify properties of heap structures. It has been shown to be quite powerful when reasoning about heaps, leading to an automated proof of total correctness for a variant of the Schorr-Waite algorithm [19] restricted to binary trees. The ideas of shape analysis have also been combined with those of separation logic [6] to quickly infer invariants for programs operating on lists.

Dynamic logic (DL) [14] extends FOL with modal operators to embed program fragments within program specifications. For example, a partial correctness Hoare triple $\{\varphi\}\ \mathtt{s}\ \{\psi\}$ can be represented as a DL formula $\varphi \rightarrow [\mathtt{s}]\psi$, where the meaning of the formula $[\mathtt{s}]\psi$ is "after executing `s`, a state may be reached which satisfies ψ". The advantage of dynamic logic is that programs and specifications co-exits in the same logic, so one needs no other encodings or translations. Perhaps the most mature program verification project based on dynamic logic is KeY [2]. The KeY project and matching logic have many common goals and similarities. Particularly, both attempt to serve as an alternative, rather than an extension, to Hoare logic, and both their current implementations rely on symbolic execution rather than weakest precondition. Even though in principle DL can employ $FOL_=$ and configurations, its current uses are still less explicit than the patterns used in matching logic, so one may still need logical encodings of configuration components such as stacks, heaps, pointer maps, etc. It could also be possible to devise a dynamic matching logic where the program fragment is pulled out from patterns and moved into modalities. However, one of the practical benefits of matching logic is that its patterns make no distinction between code or other configuration components, allowing to have transitions/rewrites between patterns in which the code is not involved at all (e.g., defining message delivery, or garbage collection).

Finally, there is a large body of work on embedding various semantic styles, including operational ones, into higher-level formalisms, and then using the latter to formally relate two or more semantics of the same language, or to prove properties about the embedded languages or about their programs. Relating semantics is practical, because one can use some semantics for some purposes and other for other purposes (e.g., executability versus verification). A representative example in this category is [22], which does that for a language almost identical to our IMP. Note, however, that there is a sharp distinction between such embedding approaches and matching logic, both in purpose and in shape. Matching logic is not an embedding of an operational semantics or of anything else; it is a program verification logic, like Hoare logic, but one inspired from operational semantics. We proved its relationship to Hoare logic in this paper to put it in context, and not to use their relationship as a basis for program verification; in fact, we advocate that one can have some benefits from using matching logic instead of Hoare logic for program verification. Nevertheless, like Hoare logic or other semantics, matching logic can also be embedded into higher-level formalisms and then use the latter to mechanize proofs of relationships to other semantics (such as the result claimed in this paper), or to even verify programs. Our current implementation itself can be regarded as such an embedding of matching logic, namely into rewrite logic.

Matching logic is new, so there is much work left to be done. The intrinsic separation available in matching logic might simplify verifying concurrent programs with shared resource access. Also, we would like to infer pattern loop invariants; since configurations in our approach are just ground terms that are being rewritten by semantic rules, and since patterns are terms over the same signature with constrained variables, we believe that narrowing and/or anti-unification can be good candidates to approach the problem. Since our matching logic verification approach makes language semantics practical, we believe that it will stimulate interest in giving formal rewrite logic semantics to various programming languages. Finally, we have paid no attention to compact the representation of patterns in user annotations; we are experimenting with a variant of our prover in which the environment is implicit, so one needs not mention it in patterns anymore.

References

1. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system. In: CASSIS'04. LNCS, vol. 3362, pp. 49–69 (2004)
2. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software: The KeY Approach, LNCS, vol. 4334. Springer (2007)
3. Berdine, J., Calcagno, C., O'Hearn, P.W.: Symbolic execution with separation logic. In: APLAS'05. LNCS, vol. 3780, pp. 52–68 (2005)

4. Clavel, M., Durán, F., Eker, S., Meseguer, J., Lincoln, P., Martí-Oliet, N., Talcott, C.: All About Maude, A High-Performance Logical Framework, LNCS, vol. 4350. Springer (2007)
5. Cohen, E., Moskal, M., Schulte, W., Tobies, S.: A practical verification methodology for concurrent programs. Tech. Rep. MSR-TR-2009-15, Microsoft Research (2009)
6. Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In: TACAS'06. pp. 287–302 (2006)
7. Distefano, D., Parkinson, M.J.: jStar: Towards practical verification for Java. In: OOPSLA'08. pp. 213–226 (2008)
8. Filliâtre, J.C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: CAV'07. LNCS, vol. 4590, pp. 173–177 (2007)
9. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: PLDI'02. pp. 234–245 (2002)
10. Floyd, R.W.: Assigning meaning to programs. In: Schwartz, J.T. (ed.) Proceedings of the Symposium on Applied Mathematics, vol. 19, pp. 19–32. AMS (1967)
11. Goguen, J., Malcolm, G.: Algebraic Semantics of Imperative Programs. MIT Press (1996)
12. Gordon, M., Collavizza, H.: Forward with Hoare. In: Reflections on the Work of C.A.R. Hoare. History of Computing Series, Springer (2010)
13. Gries, D.: The Schorr-Waite graph marking algorithm. Acta Informatica 11, 223–232 (1979)
14. Harel, D., Kozen, D., Tiuryn, J.: Dynamic logic. In: Handbook of Philosophical Logic. pp. 497–604 (1984)
15. Hoare, C.A.R.: An axiomatic basis for computer programming. CACM 12(10), 576–580 (1969)
16. Hubert, T., Marché, C.: A case study of C source code verification: The Schorr-Waite algorithm. In: SEFM'05. pp. 190–199 (2005)
17. Lahiri, S.K., Qadeer, S.: Verifying properties of well-founded linked lists. In: POPL'06. pp. 115–126 (2006)
18. Lev-Ami, T., Immerman, N., Reps, T., Sagiv, M., Srivastava, S.: Simulating reachability using first-order logic. In: CADE'05. pp. 99–115 (2005)
19. Loginov, A., Reps, T., Sagiv, M.: Automated verification of the Deutsch-Schorr-Waite tree-traversal algorithm. In: SAS'06. LNCS, vol. 4134, pp. 261–279 (2006)
20. Meseguer, J., Roşu, G.: The rewriting logic semantics project. Theoretical Computer Science 373(3), 213–237 (2007)
21. Møller, A., Schwartzbach, M.I.: The pointer assertion logic engine. SIGPLAN Not. 36(5), 221–231 (2001)
22. Nipkow, T.: Winskel is (almost) right: Towards a mechanized semantics. Formal Aspects of Computing 10(2), 171–186 (1998)
23. O'Hearn, P.W., Pym, D.J.: The logic of bunched implications. Bulletin of Symbolic Logic 5, 215–244 (1999)
24. Pavlovic, D., Smith, D.R.: Composition and refinement of behavioral specifications. In: ASE'01. pp. 157–165 (2001)
25. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS'02. pp. 55–74 (2002)
26. Roşu, G., Schulte, W.: Matching logic—Extended report. Tech. Rep. UIUCDCS-R-2009-3026, University of Illinois (2009)
27. Roşu, G., Ellison, C., Schulte, W.: From rewriting logic executable semantics to matching logic program verification. Tech. Rep. `http://hdl.handle.net/2142/13159`, UIUC (2009)
28. Roşu, G., Şerbănuţă, T.F.: An overview of the K semantic framework. Journal of Logic and Algebraic Programming 79(6), 397–434 (2010)
29. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Transactions on Programming Languages and Systems 24(3), 217–298 (2002)