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 $\{\varphi\}\ \mathtt{s}\ \{\varphi'\}$, where $\mathtt{s}$ 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 $\{\varphi\}\ \mathtt{s}\ \{\varphi'\}$ can and typically do refer to program variables that appear in $\mathtt{s}$. 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

* 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.

M. Johnson and D. Pavlovic (Eds.): AMAST 2010, LNCS 6486, pp. 142–162, 2011.

is difficult to specify recursive predicates, because they raise both theoretical (consistency) and practical (hard to automatically reason with) concerns. Solutions are either 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 $\text{FOL}_{=}$ (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 $\text{FOL}_{=}$ formula:

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

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

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