ICALP'12, LNCS 7392, pp 351-363. 2012

Towards a Unified Theory of Operational and Axiomatic Semantics [⋆]

Grigore Roşu[1,2] and Andrei Ştefănescu[1]

[1] University of Illinois at Urbana-Champaign, USA {grosu,stefane1}@illinois.edu
[2] Alexandru Ioan Cuza University, Iaşi, Romania,

Abstract. This paper presents a nine-rule *language-independent* proof system that takes an operational semantics as axioms and derives program reachability properties, including ones corresponding to Hoare triples. This eliminates the need for language-specific Hoare-style proof rules to verify programs, and, implicitly, the tedious step of proving such proof rules sound for each language separately. The key proof rule is *Circularity*, which is coinductive in nature and allows for reasoning about constructs with repetitive behaviors (e.g., loops). The generic proof system is shown sound and has been implemented in the MatchC verifier.

1 Introduction

An operational semantics defines a formal executable model of a language typically in terms of a transition relation $cfg \Rightarrow cfg'$ between program configurations, and can serve as a formal basis for language understanding, design, and implementation. On the other hand, an axiomatic semantics defines a proof system typically in terms of Hoare triples $\{\psi\}\,\mathtt{code}\,\{\psi'\}$, and can serve as a basis for program reasoning and verification. Operational semantics are well-understood and comparatively easier to define than axiomatic semantics for complex languages. More importantly, operational semantics are typically executable, and thus testable. For example, we can test them by executing the same test suites that compiler testers use, as has been done with the operational semantics of C [6]. Thus, we can build confidence in and eventually trust them.

The state-of-the-art in mechanical program verification (see, e.g., [1,9,13,15,19,27]) is to describe the trusted operational semantics in a powerful logical framework or language, say $\mathcal{L}$, and then to use the capabilities of $\mathcal{L}$ (e.g., induction) to verify programs. To avoid proving low-level and program-specific lemmas, Hoare-style proof rules (or consequences of them such as weakest-precondition or strongest-postcondition procedures) are typically also formalized and proved sound in $\mathcal{L}$ w.r.t. the given operational semantics. Despite impressive mechanical theorem proving advances in recent years, language designers still perceive the operational and axiomatic semantics as two distinct endeavors, and proving their formal relationship as a burden. With few notable exceptions, real languages are rarely given both semantics. Consequently, many program verifiers end up building upon possibly unsound axiomatic semantics.

The above lead naturally to the idea of a *unified theory* of programming, in the sense of [12], where various semantic approaches coexists with systematic relationships between them. The disadvantage of the approach in [12] is that one still needs two or more

[⋆] Full version of this paper, with proofs, available at `http://hdl.handle.net/2142/30827`.

semantics of the same language. Another type of a unified theory could be one where we need only *one* semantics of the language, the theory providing the necessary machinery to achieve the same benefits as in each individual semantics, at the same cost. In the context of operational and axiomatic semantics, such a theory would have the following properties: (1) it is as executable, testable, and simple as operational semantics, so it can be used to define sound-by-construction models of programming languages; and (2) it is as good for program reasoning and verification as axiomatic semantics, so no other semantics for verification purposes—and, implicitly, no tedious soundness proofs—are needed. Such a unified theory could be, for example, a language-independent Hoare-logic-like framework taking the operational semantics rules as axioms. To understand why this is not easy, consider the Hoare logic rule for `while` in a C-like language:

$$\frac{\mathcal{H} \vdash \{\varphi \wedge e \neq 0\}\, s\, \{\varphi\}}{\mathcal{H} \vdash \{\varphi\}\, \mathtt{while}(e)\, s\, \{\varphi \wedge e = 0\}}$$

This proof rule is far from being language-independent. It heavily relies on the C-like semantics of this particular `while` construct ($e = 0$ means e is false, $e \neq 0$ means e is true). If by mistake we replace $e \neq 0$ with $e = 1$, then we get a wrong Hoare logic. This problem is amplified by its lack of executability/testability, which is why each Hoare logic needs to be proved sound w.r.t. a trusted semantics for each language separately.

We present the first steps towards such a unified theory of operational and axiomatic semantics. Our result is a sound and language-independent proof system for *matching logic reachability rules* $\varphi \Rightarrow \varphi'$ between *patterns*. A pattern φ specifies a set of program configurations that *match* it. A rule $\varphi \Rightarrow \varphi'$ specifies reachability: configurations matching φ eventually transit to ones matching φ'. Patterns were used in [21] to define language-specific axiomatic semantics (e.g., the `while` proof rule was similar to the one above). Our new approach is much closer to a unified theory. Although we support a limited number of operational semantics styles (including the popular reduction semantics with evaluation contexts [7]), our new proof system is language-independent.

Matching logic reachability rules generalize the basic elements of both operational and axiomatic semantics. Transitions between configurations, upon which operational semantics build, are instances of one-step reachability rules. Also, Hoare triples $\{\psi\}$ `code` $\{\psi'\}$ can be viewed as reachability rules between a pattern holding `code` with constraints ψ and a pattern holding the empty code and constraints ψ'. The proof system that we propose in this paper takes an operational semantics given as a set of reachability rules, say $\mathcal{A}$ (axioms), and derives other reachability rules. The key proof rule of our system is *Circularity*, which has a coinductive nature:

$$\frac{\mathcal{A} \vdash \varphi \Rightarrow^{+} \varphi'' \qquad \mathcal{A} \cup \{\varphi \Rightarrow \varphi'\} \vdash \varphi'' \Rightarrow \varphi'}{\mathcal{A} \vdash \varphi \Rightarrow \varphi'}$$

It deductively and language-independently captures the various circular behaviors that appear in languages, due to loops, recursion, etc. Circularity adds new reachability rules to $\mathcal{A}$ during the proof derivation process, which can be used in their own proof! The correctness of this proof circularity is given by the fact that progress is required to be made (indicated by $\Rightarrow^{+}$ in $\mathcal{A} \vdash \varphi \Rightarrow^{+} \varphi''$) before a circular reasoning step is allowed.

Sections 2 and 3 recall operational semantics and matching logic patterns. Sections 4 and 5 contain our novel theoretical notions and contribution, the sound proof system for matching logic reachability. Section 6 discusses MatchC, an automated program verifier based on our proof system. Section 7 discusses related and future work, and concludes.

ICALP'12, LNCS 7392, pp 351-363. 2012

IMP language syntax

$$\begin{array}{rcl}
\mathit{PVar} & ::= & \text{program variables} \\
\mathit{Exp} & ::= & \mathit{PVar} \mid \mathit{Int} \mid \mathit{Exp}\ \texttt{op}\ \mathit{Exp} \\
\mathit{Stmt} & ::= & \texttt{skip} \mid \mathit{PVar} := \mathit{Exp} \mid \mathit{Stmt};\mathit{Stmt} \\
 & \mid & \texttt{if}(\mathit{Exp})\ \mathit{Stmt}\ \texttt{else}\ \mathit{Stmt} \\
 & \mid & \texttt{while}(\mathit{Exp})\ \mathit{Stmt}
\end{array}$$

IMP evaluation contexts syntax

$$\begin{array}{rcl}
\mathit{Context} & ::= & \blacksquare \\
 & \mid & \langle \mathit{Context},\ \mathit{State} \rangle \\
 & \mid & \mathit{Context}\ \texttt{op}\ \mathit{Exp} \mid \mathit{Int}\ \texttt{op}\ \mathit{Context} \\
 & \mid & \mathit{PVar} := \mathit{Context} \mid \mathit{Context};\mathit{Stmt} \\
 & \mid & \texttt{if}(\mathit{Context})\ \mathit{Stmt}\ \texttt{else}\ \mathit{Stmt}
\end{array}$$

IMP operational semantics

$$\begin{array}{ll}
\textbf{lookup} & \langle C,\ \sigma \rangle[x] \Rightarrow \langle C,\ \sigma \rangle[\sigma(x)] \\
\textbf{op} & i_1\ \texttt{op}\ i_2 \Rightarrow i_1\ \mathit{op}_{\mathit{Int}}\ i_2 \\
\textbf{asgn} & \langle C,\ \sigma \rangle[x := i] \Rightarrow \langle C,\ \sigma[x \leftarrow i] \rangle[\texttt{skip}] \\
\textbf{seq} & \texttt{skip};s_2 \Rightarrow s_2
\end{array}
\qquad
\begin{array}{ll}
\textbf{cond}_1 & \texttt{if}(i)\ s_1\ \texttt{else}\ s_2 \Rightarrow s_1 \quad \text{if } i \neq 0 \\
\textbf{cond}_2 & \texttt{if}(0)\ s_1\ \texttt{else}\ s_2 \Rightarrow s_2 \\
\textbf{while} & \texttt{while}(e)\ s \Rightarrow \\
 & \texttt{if}(e)\ s;\texttt{while}(e)\ s\ \texttt{else}\ \texttt{skip}
\end{array}$$

Fig. 1. IMP language syntax and operational semantics based on evaluation contexts.

2 Operational Semantics, Reduction Rules, and Transition Systems

Here we recall basic notions of operational semantics, reduction rules, and transition systems, and introduce our notation and terminology for these. We do so by means of a simple imperative language, IMP. Fig. 1 shows its syntax and an operational semantics based on evaluation contexts. IMP has only integer expressions. When used as conditions of `if` and `while`, zero means false and any non-zero integer means true (like in C). Expressions are formed with integer constants, program variables, and conventional arithmetic constructs. For simplicity, we only assume a generic binary operation, `op`. IMP statements are the variable assignment, `if`, `while` and sequential composition.

Various operational semantics styles define programming languages (or calculi, or systems, etc.) as sets of rewrite or reduction rules of the form "$l \Rightarrow r$ if b", where l and r are program configurations with variables constrained by the boolean condition b. One of the most popular such operational approaches is reduction semantics with evaluation contexts [7], with rules "$C[t] \Rightarrow C'[t']$ if b", where C is the evaluation context which reduces to C' (typically $C = C'$), t is the redex which reduces to t', and b is a side condition. Another approach is the chemical abstract machine [4], where l is a chemical solution that reacts into r under condition b. The $\mathbb{K}$ framework [23] is another, based on plain (no evaluation contexts) rewrite rules. Several large languages have been given such semantics, including C [6] (whose definition has about 1200 such rules).

Here we chose to define IMP using reduction semantics with evaluation contexts. Note, however, that our subsequent results work with any of the aforementioned operational approaches. The program configurations of IMP are pairs $\langle \texttt{code},\ \sigma \rangle$, where `code` is a program fragment and σ is a state term mapping program variables into integers. As usual, we assume appropriate definitions for the integer and map domains available, together with associated operations like arithmetic operations ($i_1\ \mathit{op}_{\mathit{Int}}\ i_2$, etc.) on the integers and lookup ($\sigma(x)$) or update ($\sigma[x \leftarrow i]$) on the maps.

The IMP definition in Fig. 1 consists of seven reduction rule schemas between program configurations, which make use of first-order variables: σ is a variable of sort *State*; x is a variable of sort *PVar*; i, i_1, i_2 are variables of sort *Int*; e is a variable of sort *Exp*; s, s_1, s_2 are variables of sort *Stmt*. A rule mentions a context (containing a code context and a state) and a redex which together form a configuration, and reduces the said configuration by rewriting the redex and possibly the context. As notational shortcut, a context is not mentioned if not used; e.g., the rule **op** stands in fact for the rule $\langle C,\ \sigma\rangle[i_1 \,\mathsf{op}\, i_2] \Rightarrow \langle C,\ \sigma\rangle[i_1\, op_{Int}\, i_2]$. The code context meta-variable C allows one to instantiate a schema into reduction rules, one for each valid redex of each code fragment. For example, with C set to $x := \blacksquare \,\mathsf{op}\, y$, the **op** rule schema becomes the rule $\langle x := (i_1 \,\mathsf{op}\, i_2)\,\mathsf{op}\, y,\ \sigma\rangle \Rightarrow \langle x := (i_1\, op_{Int}\, i_2)\,\mathsf{op}\, y,\ \sigma\rangle$.

We can therefore regard the operational semantics of IMP above as a set of reduction rules of the form "$l \Rightarrow r$ if b", where l and r are configurations with variables constrained by the boolean condition b. The subsequent results work with such sets of reduction rules and are agnostic to the particular underlying operational semantics style.

Let $\mathcal{S}$ (for "semantics") be a set of reduction rules like above, and let Σ be the underlying signature; also, let *Cfg* be a distinguished sort of Σ (for "configurations"). $\mathcal{S}$ yields a transition system on any Σ-algebra/model $\mathcal{T}$, no matter whether $\mathcal{T}$ is a term model or not. Let us fix an arbitrary model $\mathcal{T}$, which we may call a *configuration model*; as usual, $\mathcal{T}_{Cfg}$ denotes the elements of $\mathcal{T}$ of sort *Cfg*, which we call *configurations*:

Definition 1. *$\mathcal{S}$ induces a **transition system** $(\mathcal{T}, \Rightarrow^{\mathcal{T}}_{\mathcal{S}})$ as follows: $\gamma \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma'$ for some $\gamma, \gamma' \in \mathcal{T}_{Cfg}$ iff there is some rule "$l \Rightarrow r$ if b" in $\mathcal{S}$ and some $\rho : Var \to \mathcal{T}$ such that $\rho(l) = \gamma$, $\rho(r) = \gamma'$ and $\rho(b)$ holds (Var is the set of variables appearing in rules in $\mathcal{S}$ and we used the same ρ for its homomorphic extension to terms l, r and predicates b).*

$(\mathcal{T}, \Rightarrow^{\mathcal{T}}_{\mathcal{S}})$ is a conventional transition system, i.e., a set together with a binary relation on it (in fact, $\Rightarrow^{\mathcal{T}}_{\mathcal{S}} \subseteq \mathcal{T}_{Cfg} \times \mathcal{T}_{Cfg}$), and captures precisely how the language defined by $\mathcal{S}$ operates. We use it in Section 5 to define and prove the soundness of our proof system.

3 Matching Logic Patterns

Here we recall the notion of a matching logic pattern from [21]. Note that this section is the only overlap between [21] and this paper. The objective there was to use patterns as a state specification logic to give language-specific axiomatic (non-operational) semantics. The approach and proof system in this paper (Sections 4 and 5) are quite different: they are language-independent and use the operational semantics rules as axioms.

We assume the reader is familiar with basic concepts of algebraic specification and first-order logic. Given an algebraic signature Σ, we let $\mathcal{T}_\Sigma$ denote the initial Σ-algebra of ground terms (i.e., terms without variables) and let $\mathcal{T}_\Sigma(X)$ denote the free Σ-algebra of terms with variables in X. $\mathcal{T}_{\Sigma,s}(X)$ denotes the set of Σ-terms of sort s. These notions extend to algebraic specifications. Many mathematical and computing structures can be defined as initial Σ-algebras: boolean algebras, natural/integer/rational numbers, monoids, groups, rings, lists, sets, bags (or multisets), mappings, trees, queues, stacks, etc. CASL [17] and Maude [5] use first-order and algebraic specifications as underlying semantic infrastructure; we refer the reader to [5, 17] for examples. Here we only need maps, to represent program states. We use the notation $Map_{PVar,Int}$ for the sort corresponding to maps taking program variables into integers. We use an infix "$\mapsto$" for map entries and (an associative and commutative) comma "," to separate them.

Matching logic is parametric in configurations, or more precisely in a configuration model. Next, we discuss the configuration signature of IMP, noting that different languages or calculi have different signatures. The same machinery works for all.

ICALP'12, LNCS 7392, pp 351-363. 2012

$$\boxed{\begin{array}{rl} \mathit{PVar} ::= & \text{IMP identifiers} \\ \mathit{Int} ::= & \text{integer numbers} \\ \mathit{Syntax} ::= & \text{IMP syntax} \\ \mathit{State} ::= & \mathit{Map}_{\mathit{PVar},\,\mathit{Int}} \\ \mathit{Cfg} ::= & \langle \mathit{Syntax},\ \mathit{State} \rangle \end{array}}$$

Fig. 2. IMP configurations

Fig. 2 shows the configuration syntax of IMP. The sort *Syntax* is a generic sort for "code". Thus, terms of sort *Syntax* correspond to program fragments. States are terms of sort *State*, mapping program variables to integers. A program configuration is a term $\langle \texttt{code}, \sigma \rangle$ of sort *Cfg*, with `code` a term of sort *Syntax* and σ of sort *State*.

Let Σ be the algebraic signature associated to some desired configuration syntax. Then a Σ-algebra gives a configuration *model*, namely a universe of concrete configurations. From here on we assume that Σ is a fixed signature and $\mathcal{T}$ a fixed configuration model. Note that $\mathcal{T}$ can be quite large (including models of integers, maps, etc.). We assume that Σ has a distinguished sort *Cfg* and *Var* is a sort-wise infinite set of variables.

Definition 2. *Matching logic extends the syntax of first order logic with equality (abbreviated FOL) by adding Σ-terms with variables, called **basic patterns**, as formulae:*

$$\varphi ::= \ldots \textit{ conventional FOL syntax } \mid \mathcal{T}_{\Sigma,\mathit{Cfg}}(\mathit{Var})$$

*Matching logic formulae are also called **patterns**.*

Let $\psi, \psi_1, \psi', \ldots$ range over conventional FOL formulae (without patterns), $\pi, \pi_1, \pi', \ldots$ over basic patterns, and $\varphi, \varphi_1, \varphi', \ldots$ over any patterns. Matching logic satisfaction is *(pattern) matching* in $\mathcal{T}$. The satisfaction of the FOL constructs is standard. We extend FOL's valuations to include a $\mathcal{T}$ configuration, to be used for matching basic patterns:

Definition 3. *We define the relation $(\gamma, \rho) \models \varphi$ over configurations $\gamma \in \mathcal{T}_{\mathit{Cfg}}$, valuations $\rho : \mathit{Var} \to \mathcal{T}$ and patterns φ as follows (among the FOL constructs, we only show $\exists$):*

$(\gamma, \rho) \models \exists X\, \varphi$ *iff* $(\gamma, \rho') \models \varphi$ *for some* $\rho' : \mathit{Var} \to \mathcal{T}$ *with* $\rho'(y) = \rho(y)$ *for all* $y \in \mathit{Var} \backslash X$

$(\gamma, \rho) \models \pi$ *iff* $\gamma = \rho(\pi)$ *where* $\pi \in \mathcal{T}_{\Sigma,\mathit{Cfg}}(\mathit{Var})$

We write $\models \varphi$ when $(\gamma, \rho) \models \varphi$ for all $\gamma \in \mathcal{T}_{\mathit{Cfg}}$ and all $\rho : \mathit{Var} \to \mathcal{T}$.

The pattern below matches the IMP configurations holding the code computing the sum of the natural numbers from 1 to `n`, and the state mapping the program variables `s`, `n` into integers s, n, such that n is positive. We use `typewriter` fonts for program variables and *italic* fonts for mathematical variables.

$$\exists s\, (\langle\ \texttt{s:=0; while(n>0)(s:=s+n; n:=n-1)},\ (\texttt{s} \mapsto s,\ \texttt{n} \mapsto n)\ \rangle \wedge n \geq_{\mathit{Int}} 0)$$

Note that s is existentially quantified, while n is not. That means that n can be further constrained if the pattern above is put in some larger context. Similarly, the pattern

$$\langle \texttt{skip},\ (\texttt{s} \mapsto n *_{\mathit{Int}} (n +_{\mathit{Int}} 1) /_{\mathit{Int}} 2,\ \texttt{n} \mapsto 0) \rangle$$

is satisfied (with the same ρ, that is, the same n) by all the final configurations reachable (in IMP's transition system) from the configurations specified by the previous pattern.

Next, we show how matching logic formulae can be translated into FOL formulae, so that its satisfaction becomes FOL satisfaction in the model of configurations, $\mathcal{T}$.

Definition 4. *Let $\square$ be a fresh Cfg variable. For a pattern φ, let $\varphi^{\square}$ be the FOL formula replacing basic patterns $\pi \in \mathcal{T}_{\Sigma,\mathit{Cfg}}(\mathit{Var})$ with equalities $\square = \pi$. If $\rho : \mathit{Var} \to \mathcal{T}$ and $\gamma \in \mathcal{T}_{\mathit{Cfg}}$ then let $\rho^{\gamma} : \mathit{Var} \cup \{\square\} \to \mathcal{T}$ be the mapping $\rho^{\gamma}(x) = \rho(x)$ for $x \in \mathit{Var}$ and $\rho^{\gamma}(\square) = \gamma$.*

With the notation in Definition 4, $(\gamma, \rho) \models \varphi$ iff $\rho^{\gamma} \models_{\mathrm{FOL}} \varphi^{\square}$, and $\models \varphi$ iff $\mathcal{T} \models_{\mathrm{FOL}} \varphi^{\square}$. Therefore, matching logic is a methodological fragment of the FOL theory of $\mathcal{T}$. Thus, we can actually use conventional theorem provers or proof assistants for pattern reasoning.

4 Matching Logic Reachability

Patterns were used in [21] to specify state properties in axiomatic semantics. Unfortunately, that approach shares a major disadvantage with other axiomatic approaches: the target language needs to be given a new semantics. Axiomatic semantics are less intuitive than operational semantics, are not easily executable, and thus are hard to test and need to be proved sound. What we want is *one* formal semantics of a language, which should be both executable and suitable for program verification. In this section we introduce the notion of *matching logic reachability*, and show that it captures operational semantics and can be used to specify reachability properties about programs. Assume an arbitrary but fixed configuration signature Σ and a model $\mathcal{T}$, like in Sections 2 and 3.

Definition 5. *A (matching logic)* ***reachability rule*** *is a pair $\varphi \Rightarrow \varphi'$, where φ, called the* ***left-hand side (LHS)****, and φ', called the* ***right-hand side (RHS)****, are matching logic patterns (which can have free variables). A (matching logic)* ***reachability system*** *is a set of reachability rules. A reachability system $\mathcal{S}$ induces a* ***transition system*** *$(\mathcal{T}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$ on the configuration model $\mathcal{T}$, as follows: $\gamma \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma'$ for some $\gamma, \gamma' \in \mathcal{T}_{Cfg}$ iff there is some rule $\varphi \Rightarrow \varphi'$ in $\mathcal{S}$ and some $\rho : Var \to \mathcal{T}$ such that $(\gamma, \rho) \models \varphi$ and $(\gamma', \rho) \models \varphi'$. Configuration $\gamma \in \mathcal{T}_{Cfg}$* ***terminates*** *in $(\mathcal{T}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$ iff there is no infinite $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$-sequence starting with γ. A rule $\varphi \Rightarrow \varphi'$ is* ***well-defined*** *iff for any $\gamma \in \mathcal{T}_{Cfg}$ and $\rho : Var \to \mathcal{T}$ with $(\gamma, \rho) \models \varphi$, there is some $\gamma' \in \mathcal{T}_{Cfg}$ with $(\gamma', \rho) \models \varphi'$. The reachability system $\mathcal{S}$ is* ***well-defined*** *iff each rule is well-defined, and is* ***deterministic*** *iff $(\mathcal{T}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$ is deterministic.*

As mentioned in Section 2, various operational semantics styles define languages as sets of rules "$l \Rightarrow r$ if b", where l and r are *Cfg* terms with variables constrained by boolean condition b (a predicate over the variables in l and r). These reduction rules are just special matching logic reachability rules: "$l \Rightarrow r$ if b" can be viewed as the reachability rule $l \wedge b \Rightarrow r$, as both specify the same transitions $\gamma \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma'$ between configurations $\gamma, \gamma' \in \mathcal{T}_{Cfg}$. This is because Definition 3 implies that $(\gamma, \rho) \models l \wedge b$ and $(\gamma', \rho) \models r$ (like in Definition 5) iff $\rho(l) = \gamma$, $\rho(r) = \gamma'$ and $\rho(b)$ holds (like in Definition 1). Note that well-definedness makes sense in general (since, e.g., φ' can be *false*), but that matching logic rules of the form $l \wedge b \Rightarrow r$ are always well-defined (pick γ' to be $\rho(r)$).

Hence, any operational semantics defined using reduction rules like above *is* a particular matching logic reachability system. If we relax the meaning of $\varphi \Rightarrow \varphi'$ from "one step" to "zero, one or more steps", then reachability rules can in fact express arbitrary reachability program properties. Consider for example the IMP code fragment "`s:=0; while(n>0)(s:=s+n; n:=n-1)`", say `SUM`. We can express its semantics as:

$\langle \texttt{SUM}, (\texttt{s} \mapsto s, \texttt{n} \mapsto n)\rangle \wedge n \geq_{Int} 0 \Rightarrow \langle \texttt{skip}, (\texttt{s} \mapsto n *_{Int} (n +_{Int} 1)/_{Int} 2, \texttt{n} \mapsto 0)\rangle$

This says that any configuration γ holding `SUM` and some state binding program variables `s` and `n` to integers s and respectively positive n, eventually transits to a configuration γ' whose code is consumed, `s` is bound to the sum of numbers up to n and `n` is 0. Note that $s, n \in Var$ are free logical variables in this rule, so they are instantiated the same way in γ and γ', while `s` and `n` are program variables, that is, constants of sort *PVar*.

As shown in [25], to *any* IMP Hoare triple $\{\psi\}\,\texttt{code}\,\{\psi'\}$ we can associate the following reachability rule: $\exists X_{\texttt{code}}(\langle \texttt{code}, \sigma_{X_{\texttt{code}}}\rangle \wedge \psi_X) \Rightarrow \exists X_{\texttt{code}}(\langle \texttt{skip}, \sigma_{X_{\texttt{code}}}\rangle \wedge \psi'_X)$, where X is a set containing a logical integer variable x for each variable `x` appearing in the Hoare triple, $X_{\texttt{code}} \subseteq X$ is the subset corresponding to program variables appearing in `code`, $\sigma_{X_{\texttt{code}}}$ binds each program variable `x` to its logical variable x, and ψ_X, ψ'_X are the formulae obtained from ψ, ψ' by replacing each variable `x` with its corresponding logical variable x and each arithmetic operation `op` with its corresponding domain operation op_{Int}. As an example, consider the Hoare triple specifying the semantics of `SUM` above, namely $\{\texttt{n} = \texttt{oldn} \wedge \texttt{n} \geq 0\}$ `SUM` $\{\texttt{s = oldn*(oldn+1)/2} \wedge \texttt{n} = 0\}$. The `oldn` variable is needed to remember the initial value of `n`. Hoare logic makes no theoretical distinction between program and logical variables, nor between program expression constructs and logical expression constructs. The corresponding matching logic reachability rule is

$$\begin{array}{l} \quad \exists s, n\,(\langle \texttt{SUM},\ (\texttt{s} \mapsto s,\ \texttt{n} \mapsto n)\rangle \wedge n = oldn \wedge n \geq_{Int} 0) \\ \Rightarrow \exists s, n\,(\langle \texttt{skip},\ (\texttt{s} \mapsto s,\ \texttt{n} \mapsto n)\rangle \wedge s = oldn *_{Int} (oldn +_{Int} 1)/_{Int}2 \wedge n = 0) \end{array}$$

While this reachability rule mechanically derived from the Hoare triple is more involved than the one we originally proposed, it is not hard to see that they specify the same pairs of configurations γ, γ'. The proof system in Section 5 allows one to formally show them equivalent. Therefore, in the case of IMP, we can use reachability rules as an alternative to Hoare triples for specifying program properties. Note that reachability rules are strictly more expressive than Hoare triples, as they allow any code in their RHS patterns, not only `skip`. Replacing Hoare logic reasoning by matching logic reachability reasoning using translations like above is discouraged, as one would be required to still provide a Hoare logic for the target language. A strong point of our approach is that one does *not* have to go through this tedious step. We implemented the proof system in Section 5 and verified many programs (see Section 6) with it, using only the operational semantic rules, without having to prove any Hoare logic proof rules as lemmas. The reason we show this translation here is only to argue that the matching logic reachability rules are expressive.

Next, we define the semantic validity of matching logic reachability rules. Recall the semantic validity (i.e., partial correctness) of Hoare triples: $\models \{\psi\}$ `code` $\{\psi'\}$ iff for all states s, if $s \models \psi$ and `code` executed in state s terminates in state s', then $s' \models \psi'$. This elegant definition has the luxury of relying on another (typically operational or denotational) semantics of the language, which provides the notions of "execution", "termination" and "state". Here all these happen in the same semantics, given as a reachability system. The closest matching logic element to a "state" is a ground configuration in $\mathcal{T}_{Cfg}$, which also includes the code. Since the transition system $(\mathcal{T}, \Rightarrow^{\mathcal{T}}_{\mathcal{S}})$ is meant to give all the operational behaviors of the defined language, we introduce the following

Definition 6. *Let $\mathcal{S}$ be a reachability system and $\varphi \Rightarrow \varphi'$ a reachability rule. We define $\mathcal{S} \models \varphi \Rightarrow \varphi'$ iff for all $\gamma \in \mathcal{T}_{Cfg}$ and $\rho : Var \to \mathcal{T}$ such that γ terminates in $(\mathcal{T}, \Rightarrow^{\mathcal{T}}_{\mathcal{S}})$ and $(\gamma, \rho) \models \varphi$, there exists some $\gamma' \in \mathcal{T}_{Cfg}$ such that $\gamma \Rightarrow^{\star\mathcal{T}}_{\mathcal{S}} \gamma'$ and $(\gamma', \rho) \models \varphi'$.*

Intuitively, $\mathcal{S} \models \varphi \Rightarrow \varphi'$ specifies reachability, in the sense that "any terminating ground configuration that matches φ transits, on some possible execution path, to a configuration that matches φ'". If $\mathcal{S}$ is deterministic (Definition 5), then "some path" becomes equivalent to "all paths", and thus $\varphi \Rightarrow \varphi'$ captures partial correctness.

Note that, if $\mathcal{S}$ is well-defined, then $\mathcal{S} \models \varphi \Rightarrow \varphi'$ holds for all $\varphi \Rightarrow \varphi' \in \mathcal{S}$ (well-defined axioms are semantically valid). Also, as already mentioned, φ' does not need to have an empty (`skip`) code cell. If it does, then so does γ' in the definition above, and, in the case of IMP, γ' is unique and thus we recover the Hoare validity as a special case.

Taking $\mathcal{S}$ to be the operational semantics of IMP in Section 2, $\mathcal{S} \models \varphi \Rightarrow \varphi'$ can be proved for the two example reachability rules $\varphi \Rightarrow \varphi'$ for `SUM` in this section. Unfortunately, such proofs are tedious, involving low-level details about the IMP transition

Rules of operational nature

Reflexivity :
$$\frac{\cdot}{\mathcal{A} \vdash \varphi \Rightarrow \varphi}$$

Axiom :
$$\frac{\varphi \Rightarrow \varphi' \in \mathcal{A}}{\mathcal{A} \vdash \varphi \Rightarrow \varphi'}$$

Substitution :
$$\frac{\mathcal{A} \vdash \varphi \Rightarrow \varphi' \qquad \theta : Var \to \mathcal{T}_\Sigma(Var)}{\mathcal{A} \vdash \theta(\varphi) \Rightarrow \theta(\varphi')}$$

Transitivity :
$$\frac{\mathcal{A} \vdash \varphi_1 \Rightarrow \varphi_2 \qquad \mathcal{A} \vdash \varphi_2 \Rightarrow \varphi_3}{\mathcal{A} \vdash \varphi_1 \Rightarrow \varphi_3}$$

Rules of deductive nature

Case Analysis :
$$\frac{\mathcal{A} \vdash \varphi_1 \Rightarrow \varphi \qquad \mathcal{A} \vdash \varphi_2 \Rightarrow \varphi}{\mathcal{A} \vdash \varphi_1 \vee \varphi_2 \Rightarrow \varphi}$$

Logic Framing :
$$\frac{\mathcal{A} \vdash \varphi \Rightarrow \varphi' \qquad \psi \text{ is a (patternless) FOL formula}}{\mathcal{A} \vdash \varphi \wedge \psi \Rightarrow \varphi' \wedge \psi}$$

Consequence :
$$\frac{\models \varphi_1 \to \varphi_1' \qquad \mathcal{A} \vdash \varphi_1' \Rightarrow \varphi_2' \qquad \models \varphi_2' \to \varphi_2}{\mathcal{A} \vdash \varphi_1 \Rightarrow \varphi_2}$$

Abstraction :
$$\frac{\mathcal{A} \vdash \varphi \Rightarrow \varphi' \qquad X \cap FreeVars(\varphi') = \emptyset}{\mathcal{A} \vdash \exists X\, \varphi \Rightarrow \varphi'}$$

Rule for circular behavior

Circularity :
$$\frac{\mathcal{A} \vdash \varphi \Rightarrow^+ \varphi'' \qquad \mathcal{A} \cup \{\varphi \Rightarrow \varphi'\} \vdash \varphi'' \Rightarrow \varphi'}{\mathcal{A} \vdash \varphi \Rightarrow \varphi'}$$

Fig. 3. Matching logic reachability proof system.

system and induction. What we want is an abstract proof system for deriving matching logic reachability rules, which does not refer to the low-level transition system.

5 Sound Proof System

Fig. 3 shows our nine-rule language-independent proof system for deriving matching logic reachability rules. We start with a set of reachability rules representing an operational semantics of the target language, and then either "execute" programs or derive program properties in a generic manner, without relying on the specifics of the target language except for using its operational semantics rules as axioms. In particular, no auxiliary lemmas corresponding to Hoare logic rules are proved or needed. Initially, $\mathcal{A}$ contains the operational semantics of the target language. The first group of rules (Reflexivity, Axiom, Substitution, Transitivity) has an operational nature and derives concrete and (linear) symbolic executions; any executable semantic framework has similar rules (see, e.g., rewriting logic [16]). The second group of rules (Case Analysis, Logic Framing, Consequence and Abstraction) has a more deductive nature and is inspired from the subset of language-independent rules of Hoare logic [11]. The first group combined with Case Analysis and Logic Framing enables the derivation of all symbolic execution paths. The Circularity proof rule is new and captures the various circular behaviors that appear in languages, due to loops, recursion, jumps, etc.

Let $\mathcal{A} \vdash \varphi \Rightarrow^+ \varphi'$ be the derivation relation obtained by dropping the Reflexivity rule from the proof system in Fig. 3. $\mathcal{A} \vdash \varphi \Rightarrow^+ \varphi'$ says that a configuration satisfying φ can transit in one or more operational semantics steps to one satisfying φ'. The Circularity rule in Fig. 3 says that we can derive the sequent $\mathcal{A} \vdash \varphi \Rightarrow \varphi'$ whenever we can derive the rule $\varphi \Rightarrow \varphi'$ by starting with one or more steps in $\mathcal{A}$ and continuing with steps which can involve both rules from $\mathcal{A}$ and the rule to be proved itself, $\varphi \Rightarrow \varphi'$. The first step can for example be a loop unrolling step in the case of loops, or a function invocation step in the case of recursive functions, etc. The use of the claimed properties in their own proofs in Circularity is reminiscent of *circular coinduction* [22]. Like in circular coinduction, where the claimed properties can only be used in some special contexts, Circularity also disallows their unrestricted use: it only allows them to be guarded by a trusted, operational step. It would actually be unsound to drop the operational-step-guard requirement: for example, if $\mathcal{A}$ contained $\varphi_1 \Rightarrow \varphi_2$ then $\varphi_2 \Rightarrow \varphi_1$ could be "proved" in a two-step transitivity, using itself, the rule in $\mathcal{A}$ and then itself again.

Theorem 1. ***Soundness*** *(see [26] for the proof) Let $\mathcal{S}$ be a well-defined matching logic reachability system (typically corresponding to an operational semantics), and let $\mathcal{S} \vdash \varphi \Rightarrow \varphi'$ be a sequent derived with the proof system in Fig. 3. Then $\mathcal{S} \models \varphi \Rightarrow \varphi'$.*

Hence, proof derivations are sound w.r.t. the (transition system generated by the) operational semantics. The well-definedness requirement is acceptable (operational semantics satisfy it) and needed (otherwise not even the axioms satisfy $\mathcal{S} \models \varphi \Rightarrow \varphi'$).

We next illustrate our proof system by means of examples. The proof below may seem low level when compared to the similar proof done using Hoare logic. However, note that it is quite mechanical, the user only having to provide the invariant (φ_{inv}). The rest is automatic and consists of applying the operational reduction rules whenever they match, except for the circularities which are given priority; when the redex is an `if`, a Case Analysis is applied. Our current MatchC implementation can prove it automatically, as well as much more complex programs (see Section 6). Although the paper Hoare logic proofs for simple languages like IMP may look more compact, note that in general they make assumptions which need to be addressed in implementations, such as that expressions do not have side effects, or that substitution is available and atomic, etc.

Consider the `SUM` code (Section 4) "`s:=0; while(n>0)(s:=s+n; n:=n-1)`", and the property given as a matching logic reachability rule, say $\mu^1_{\texttt{SUM}} \equiv (\varphi_{\text{LHS}} \Rightarrow \varphi_{\text{RHS}})$:

$\langle \texttt{SUM},\ (\texttt{s} \mapsto s,\ \texttt{n} \mapsto n)\rangle \wedge n \geq_{Int} 0 \Rightarrow \langle \texttt{skip},\ (\texttt{s} \mapsto n *_{Int} (n +_{Int} 1)/_{Int}2,\ \texttt{n} \mapsto 0)\rangle$

Let us formally derive this property using the proof system in Fig. 3. Let $\mathcal{S}$ be the operational semantics of IMP in Fig. 1 and let φ_{inv} be the pattern

$$\langle \texttt{LOOP},\ (\texttt{s} \mapsto (n -_{Int} n') *_{Int} (n +_{Int} n' +_{Int} 1)/_{Int}2,\ \texttt{n} \mapsto n')\rangle \ \wedge\ n' \geq_{Int} 0$$

where `LOOP` is "`while (n>0) (s := s+n; n := n-1)`". We derive $\mathcal{S} \vdash \mu^1_{\texttt{SUM}}$ by Transitivity with $\mu_1 \equiv (\varphi_{\text{LHS}} \Rightarrow \exists n' \varphi_{\text{inv}})$ and $\mu_2 \equiv (\exists n' \varphi_{\text{inv}} \Rightarrow \varphi_{\text{RHS}})$. By Axiom **asgn** (Fig. 1, within the `SUM` context) followed by Substitution with $\theta(\sigma) = (\texttt{s} \mapsto s,\ \texttt{n} \mapsto n)$, $\theta(x) = \texttt{s}$ and $\theta(i) = 0$ followed by Logic Framing with $n \geq_{Int} 0$, we derive $\varphi_{\text{LHS}} \Rightarrow \langle \texttt{skip;LOOP},\ (\texttt{s} \mapsto 0,\ \texttt{n} \mapsto n)\rangle \ \wedge\ n \geq_{Int} 0$. This "operational" sequence of Axiom, Substitution and Logic Framing is quite common; we abbreviate it ASLF. Further, by ASLF with **seq** and Transitivity, we derive $\varphi_{\text{LHS}} \Rightarrow \langle \texttt{LOOP},\ (\texttt{s} \mapsto s,\ \texttt{n} \mapsto n)\rangle \wedge n \geq_{Int} 0$. $\mathcal{S} \vdash \mu_1$ now follows by Consequence. We derive $\mathcal{S} \vdash \mu_2$ by Circularity with $\mathcal{S} \vdash \exists n' \varphi_{\text{inv}} \Rightarrow^+ \varphi_{\texttt{if}}$ and $\mathcal{S} \cup \{\mu_2\} \vdash \varphi_{\texttt{if}} \Rightarrow \varphi_{\text{RHS}}$, where $\varphi_{\texttt{if}}$ is the formula obtained from φ_{inv} replacing its code with "`if (n>0) (s := s+n; n := n-1; LOOP) else skip`". ASLF (**while**) followed by Abstraction derive $\mathcal{S} \vdash \exists n' \varphi_{\text{inv}} \Rightarrow^+ \varphi_{\texttt{if}}$. For the other, we use Case Analysis with $\varphi_{\texttt{if}} \wedge n' \leq_{Int} 0$ and $\varphi_{\texttt{if}} \wedge n' >_{Int} 0$. ASLF (**lookup**$_{\texttt{n}}$, **op**$_{>}$, **cond**$_2$) together with some Transitivity and Consequence steps derive $\mathcal{S} \cup \{\mu_2\} \vdash \varphi_{\texttt{if}} \wedge n' \leq_{Int} 0 \Rightarrow \varphi_{\text{RHS}}$ (μ_2 is not needed in this derivation). Similarly, ASLF (**lookup**$_{\texttt{n}}$, **op**$_{>}$, **cond**$_1$, **lookup**$_{\texttt{n}}$, **lookup**$_{\texttt{s}}$, **op**$_{+}$, **asgn**, **seq**, **lookup**$_{\texttt{n}}$, **op**$_{-}$, **asgn**, **seq**, and μ_2) together with Transitivity and Consequence steps derive $\mathcal{S} \cup \{\mu_2\} \vdash \varphi_{\texttt{if}} \wedge n' >_{Int} 0 \Rightarrow \varphi_{\text{RHS}}$. This time μ_2 is needed and it is interesting to note how. After applying all the steps above and the `LOOP` fragment of code is reached again, the pattern characterizing the configuration is

$$\langle \texttt{LOOP},\ (\texttt{s} \mapsto (n -_{Int} n') *_{Int} (n +_{Int} n' +_{Int} 1)/_{Int}2 +_{Int} n',\ \texttt{n} \mapsto n' -_{Int} 1)\rangle \ \wedge\ n' >_{Int} 0$$

The circularity μ_2 can now be applied, via Consequence and Transitivity, because this formula implies $\exists n' \varphi_{\text{inv}}$ (indeed, pick the existentially quantified n' to be $n' -_{Int} 1$).

We can similarly derive the other reachability rule for SUM in Section 4, say $\mu^2_{\texttt{SUM}}$. Instead, let us prove the stronger result that the two matching logic reachability rules are equivalent, that is, that $\mu^1_{\texttt{SUM}} \vdash \mu^2_{\texttt{SUM}}$ and $\mu^2_{\texttt{SUM}} \vdash \mu^1_{\texttt{SUM}}$. Using conventional FOL reasoning and the Consequence proof rule, one can show $\mu^2_{\texttt{SUM}}$ equivalent to the reachability rule

$$\exists s\, (\langle \texttt{SUM}, (\texttt{s} \mapsto s, \texttt{n} \mapsto oldn)\rangle \wedge oldn \geq_{Int} 0) \Rightarrow \langle \texttt{skip}, (\texttt{s} \mapsto oldn *_{Int} (oldn +_{Int} 1) /_{Int} 2, \texttt{n} \mapsto 0)\rangle$$

The equivalence to $\mu^1_{\texttt{SUM}}$ now follows by applying the Substitution rule with $oldn \mapsto n$ and Consequence, and, respectively, Substitution with $n \mapsto oldn$ and Abstraction.

In the sister paper [25] (dedicated to completeness), we show that any property derived using IMP's Hoare logic proof system can also be derived using our proof system in Fig. 5, of course modulo the representation of Hoare triples as matching logic reachability rules described in Section 4. For example, a Hoare logic proof step for `while` is translated in our proof system into an Axiom step (with **while** in Fig. 1), a Case Analysis (for the resulting `if` statement), a Circularity (as part of the positive case, when the `while` statement is reached again), an Abstraction (to add existential quantifiers for the logical variables added as part of the translation), and a few Transitivity steps.

6 Implementation in MatchC

A concern to a verification framework based on operational semantics is that it may not be practical, due to the amount of required user involvement or to the amount of low-level details that need to be provided. To test the practicality of matching logic reachability, we picked a fragment of C and implemented a proof-of-concept program verifier for it based on matching logic, named MatchC. Our C fragment includes functions, structures, pointers and I/O primitives. MatchC uses reachability rules for program specifications and its implementation is directly based on the proof system in Fig. 3.

MatchC has 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. Users only provide the program specifications (reachability rules) as annotations, in addition to the unavoidable formalizations of the used mathematical domains. The rest is automatic. For example, it takes MatchC less than 2 seconds to verify Schorr-Waite for full correctness. The Matching Logic web page, `http://fsl.cs.uiuc.edu/ml`, contains an online interface to run MatchC, where users can try more than 50 existing examples (or type their own).

Let $\mathcal{S}$ be the reachability system giving the language semantics, and let C be the set of reachability rules corresponding to user-provided specifications (properties that one wants to verify). MatchC derives the rules in C using the proof system in Fig. 3. It begins by applying Circularity for each rule in C and reduces the task to deriving sequents of the form $\mathcal{S} \cup C \vdash \varphi \Rightarrow \varphi'$. To prove them, it symbolically executes φ with steps from $\mathcal{S} \cup C$ searching for a formula that implies φ'. An SMT solver (Z3 [18]) is invoked to solve the side conditions of the rules. Whenever the reduction semantics rule for a conditional statement cannot apply because its condition is symbolic, a Case Analysis is applied and formula split into a disjunction. Rules in C are given priority; thus, if each loop and function is given a specification then MatchC will always terminate (Z3 cutoff is 5s).

A previous version of MatchC, based on the proof system in [21], was discussed in [24]. The new implementation based on the proof system in Fig. 3 will be presented in detail elsewhere. We here only mean to highlight the practical feasibility of our approach.

7 Conclusion, Additional Related Work, and Future Work

To our knowledge, the proof system in Fig. 3 is the first of its kind. We can now define only *one* semantics of the target language, which is operational and thus well-understood and comparatively easier to define than an axiomatic semantics. Moreover, it is testable using existing rewrite engines or functional languages incorporating pattern matching (e.g., Haskell). For example, we can test it by executing program benchmarks that compiler testers use, like in [6]. Then, we take this semantics and use it *as is* for program verification. We not only skip completely the tedious step of having to prove the relationship between an operational and an axiomatic semantics of the same language, but we can also change the language at will (or fix semantic bugs), without having to worry about doing that in two different places and maintaining the soundness proofs.

The idea of regarding a program as a specification transformer to analyze programs in a forwards-style goes back to Floyd in 1967 [8]. However, unlike ours, Floyd's rules are language-specific, not executable, and introduce quantifiers. Dynamic logic [2, 10] extends FOL with modal operators to embed program fragments within program specifications. Like in matching logic, in dynamic logic programs and specifications also coexist in the same logic. However, unlike in matching logic, one still needs to define an alternative (dynamic logic) language semantics, with language-specific proof rules.

Leroy and Grall [14] use the coinductive interpretation of a standard big-step semantics as a semantic foundation both for terminating and for non-terminating evaluation. Our approach is different in that: (1) our proof system can reason about reachability between arbitrary formulae, rather than just the evaluation of programs to values, and (2) although Circularity has a coinductive flavor, we take the inductive interpretation of our proof system and obtain soundness by appropriate guarding of the circular rules.

Symbolic execution is a popular technique for program analysis and verification, for example automatic testing and model-checking in Java Path Finder (JPF) [20], or strongest postcondition computation in separation logic [3]. The first six proof rules in Fig. 3 can be used to symbolically execute a pattern formula φ. Different techniques are used to make symbolic execution efficient and scalable for different programming languages, like path merging in JPF. Our approach is orthogonal, in that our proof system can be used to justify those techniques based on the language operational semantics.

We believe our proof system can be extended to work with SOS-style conditional reduction rules. Concurrency and non-determinism were purposely left out; these are major topics which deserve full attention. General relative completeness and total correctness also need to be addressed. Like other formal semantics, matching logic can also be embedded into higher-level formalisms and theorem provers, so that proofs of relationships to other semantics can be mechanized, and even programs verified. Ultimately, we would like to have a generic verifier taking an operational semantics as input, together with an extension allowing users to provide pattern annotations, and to yield an automated program verifier based on the proof system in Fig. 3.

Acknowledgements: We thank Santiago Escobar, Klaus Havelund, Jose Meseguer, Peter Olveczky, Madhusudan Parthasarathy, Vlad Rusu, the members of the $\mathbb{K}$ team, and the anonymous reviewers for their valuable comments on drafts of this paper. The work in this paper was supported in part by NSA contract H98230-10-C-0294, by NSF grant CCF-0916893 and by (Romanian) SMIS-CSNR 602-12516 contract no. 161/15.06.2010.

References

1. Appel, A.W.: Verified software toolchain. In: ESOP. LNCS, vol. 6602, pp. 1–17. Springer (2011)
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. LNCS, vol. 3780, pp. 52–68. Springer (2005)
4. Berry, G., Boudol, G.: The chemical abstract machine. Th. Comp. Sci. 96(1), 217–248 (1992)
5. Clavel, M., Durán, F., Eker, S., Meseguer, J., Lincoln, P., Martí-Oliet, N., Talcott, C.: All About Maude, LNCS, vol. 4350. Springer (2007)
6. Ellison, C., Rosu, G.: An executable formal semantics of C with applications. In: POPL. pp. 533–544. ACM (2012)
7. Felleisen, M., Findler, R.B., Flatt, M.: Semantics Engineering with PLT Redex. MIT (2009)
8. Floyd, R.W.: Assigning meaning to programs. In: Symposia in Applied Mathematics. vol. 19, pp. 19–32. AMS (1967)
9. George, C., Haxthausen, A.E., Hughes, S., Milne, R., Prehn, S., Pedersen, J.S.: The RAISE Development Method. BCS Practitioner Series, Prentice Hall (1995)
10. Harel, D., Kozen, D., Tiuryn, J.: Dynamic logic. In: Handbook of Philosophical Logic. pp. 497–604 (1984)
11. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM 12(10), 576–580 (1969)
12. Hoare, C.A.R., Jifeng, H.: Unifying Theories of Programming. Prentice Hall (1998)
13. Jacobs, B.: Weakest pre-condition reasoning for java programs with JML annotations. J. Log. Algebr. Program. 58(1-2), 61–88 (2004)
14. Leroy, X., Grall, H.: Coinductive big-step operational semantics. Inf. Comput. 207(2), 284–304 (2009)
15. Liu, H., Moore, J.S.: Java program verification via a JVM deep embedding in ACL2. In: TPHOLs. LNCS, vol. 3223, pp. 184–200 (2004)
16. Meseguer, J.: Conditioned rewriting logic as a united model of concurrency. Theor. Comput. Sci. 96(1), 73–155 (1992)
17. Mosses, P.D.: CASL Reference Manual, LNCS, vol. 2960. Springer (2004)
18. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS. LNCS, vol. 4963, pp. 337–340 (2008)
19. Nipkow, T.: Winskel is (almost) right: Towards a mechanized semantics textbook. Formal Aspects of Computing 10, 171–186 (1998)
20. Pasareanu, C.S., Mehlitz, P.C., Bushnell, D.H., Gundy-Burlet, K., Lowry, M.R., Person, S., Pape, M.: Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In: ISSTA. pp. 15–26. ACM (2008)
21. Rosu, G., Ellison, C., Schulte, W.: Matching logic: An alternative to Hoare/Floyd logic. In: AMAST. LNCS, vol. 6486, pp. 142–162 (2010)
22. Rosu, G., Lucanu, D.: Circular coinduction: A proof theoretical foundation. In: CALCO. LNCS, vol. 5728, pp. 127–144 (2009)
23. Rosu, G., Serbanuta, T.F.: An overview of the K semantic framework. J. Log. Algebr. Program. 79(6), 397–434 (2010)
24. Rosu, G., Stefanescu, A.: Matching logic: A new program verification approach (NIER track). In: ICSE. pp. 868–871. ACM (2011)
25. Rosu, G., Stefanescu, A.: From Hoare logic to matching logic. In: FM. To appear (2012)
26. Rosu, G., Stefanescu, A.: Towards a unified theory of operational and axiomatic semantics. Tech. Rep. `http://hdl.handle.net/2142/30827`, Univ. of Illinois (May 2012)
27. Sasse, R., Meseguer, J.: Java+ITP: A verification tool based on Hoare logic and algebraic semantics. Electr. Notes Theor. Comput. Sci. 176(4), 29–46 (2007)