Towards a Unified Theory of Operational and Axiomatic Semantics

Grigore Roşu and Andrei Ştefănescu

Department of Computer Science, University of Illinois at Urbana-Champaign
{grosu, stefane1}@illinois.edu

Abstract. This paper presents a nine-rule *language-independent* proof system that takes an operational semantics as axioms and derives program properties, including ones corresponding to Hoare triples. This eliminates the need for language-specific Hoare-style proof rules in order 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 program 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 $\{\psi\}\ \texttt{code}\ \{\psi'\}$, 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 program benchmarks that compiler testers use, as has been done with the operational semantics of C [5]. Thus, we can build confidence in and eventually trust them.

The state-of-the art in mechanical program verification (see, e.g., [1, 8, 12, 13, 17, 22]) 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 [11], where various semantic approaches coexists with systematic relationships between them. The disadvantage of the approach in [11] is that one still needs two or more

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:

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

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 reduction rules* $\varphi \Rightarrow \varphi'$ between matching logic *patterns*. A pattern φ specifies all program configurations that *match* it. A matching logic reduction rule $\varphi \Rightarrow \varphi'$ specifies *reachability*: configurations matching φ eventually reduce to ones matching φ'. Patterns were introduced in [18], where they were used as a program logic to define language-specific axiomatic semantics (e.g., the `while` proof rule was similar to the above). Thus, the approach there was far from ideal. Our new approach is much closer. Although we support a limited number of operational semantics styles (including the popular reduction semantics with evaluation contexts [6]), our new proof system is language-independent.

Matching logic reduction rules smoothly generalize the basic elements of both operational and axiomatic semantics. They generalize the transitions between configurations, upon which operational semantics build, because configurations are particular patterns (ones with no variables). They also generalize Hoare triples, because a Hoare triple $\{\psi\}\, \texttt{code}\, \{\psi'\}$ can be regarded as a particular reduction from a pattern holding `code` with constraints ψ into a pattern holding empty code and constraints ψ'. The proposed proof system allows us to start with the set of operational semantics rules as axioms, say $\mathcal{A}$, and then derive other reduction rules, in particular ones corresponding to Hoare triples. Our proof system has nine rules. Four of them tell how reductions operationally apply, and another four capture the language-independent Hoare logic proof 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 reductions to

IMP language syntax

$PVar$ ::= program variables

Exp ::= $PVar$ | Int | Exp `op` Exp

$Stmt$ ::= `skip` | $PVar := Exp$ | $Stmt; Stmt$
| `if(`Exp`)` $Stmt$ `else` $Stmt$
| `while(`Exp`)` $Stmt$

IMP evaluation contexts syntax

$Context$ ::= ■
| $\langle Context, State \rangle$
| $Context$ `op` Exp | Int `op` $Context$
| $PVar := Context$ | $Context; Stmt$
| `if(`$Context$`)` $Stmt$ `else` $Stmt$

IMP operational semantics

lookup $\langle C, \sigma \rangle[x] \Rightarrow \langle C, \sigma \rangle[\sigma(x)]$

op i_1 `op` $i_2 \Rightarrow i_1\ op_{Int}\ i_2$

asgn $\langle C, \sigma \rangle[x := i] \Rightarrow \langle C, \sigma[x \leftarrow i] \rangle$`[skip]`

seq `skip;` $s_2 \Rightarrow s_2$

cond$_1$ `if(`i`)` s_1 `else` $s_2 \Rightarrow s_1$ if $i \neq 0$

cond$_2$ `if(0)` s_1 `else` $s_2 \Rightarrow s_2$

while `while(`e`)` $s \Rightarrow$
`if(`e`)` s`; while(`e`)` s `else skip`

Fig. 1. IMP language syntax and operational semantics

$\mathcal{A}$ 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 reductions. Section 6 discusses MatchC, an automated program verifier based on our proof system. Section 7 discusses related and future work, and concludes. Appendi A contains the soundness proof of our proof system.

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 (recursively enumerable) 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 [6], with rules "$C[t] \Rightarrow C'[t']$ if b", where C is the evaluation context which reduces to C', t is the redex which reduces to t', and b is a side condition. Another approach is the chemical abstract machine [3], where l is a chemical solution that reacts into r under condition b. The $\mathbb{K}$ framework [20] is another, based on plain (no evaluation contexts) rewrite rules. Several large languages have been given such semantics, including C [5] (whose definition has about 1200 such rules).

For concreteness, here we chose to define IMP using the most popular such operational semantics style. Note, however, that our subsequent results work with any of the aforementioned operational approaches. The program configurations 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 of the integer and map domains available, together with associated operations like arithmetic operations ($i_1\ op_{Int}\ i_2$, 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 a notational shortcut, the context is not mentioned if it is neither used nor modified. The rule **op** stands in fact for the rule $\langle C,\ \sigma \rangle [i_1\ \texttt{op}\ i_2] \Rightarrow \langle C,\ \sigma \rangle [i_1\ op_{Int}\ i_2]$. 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 $x := \blacksquare\ \texttt{op}\ y$, the **op** rule schema becomes the rule $\langle x := (i_1\ \texttt{op}\ i_2)\ \texttt{op}\ y,\ \sigma \rangle \Rightarrow \langle x := (i_1\ op_{Int}\ i_2)\ \texttt{op}\ y,\ \sigma \rangle$.

We can therefore regard the operational semantics of IMP above as a (recursively enumerable) set of 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. The subsequent results in this paper work with such reduction systems in general and are agnostic to the particular operational semantics or any other method used to produce them.

Let $\mathcal{S}$ (from "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 (from "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** $(\mathcal{T}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$ as follows: $\gamma \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma'$ 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).*

$(\mathcal{T}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$ is a conventional transition system, i.e., a set together with a binary relation on it (in fact, $\Rightarrow_{\mathcal{S}}^{\mathcal{T}} \subseteq \mathcal{T}_{Cfg} \times \mathcal{T}_{Cfg}$), 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 [18]. Note that this section is the only overlap between [18] 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 [15] and Maude [4] use first-order and algebraic specifications as underlying semantic infrastructure; we refer the reader to [4, 15] 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 to integers. We use an infix "$\mapsto$" for map bindings and (an associative and commutative) comma "," to separate them.

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

Figure 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*.

PVar	::= IMP identifiers
Int	::= integer numbers
Syntax	::= IMP syntax
State	::= $Map_{PVar,\,Int}$
Cfg	::= $\langle Syntax,\ State \rangle$

Fig. 2. IMP configurations

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:*

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

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

Let ψ, ψ_1, ψ'..., range over conventional FOL formulae (no patterns), π, π_1, π'..., over basic patterns, and φ, φ_1, φ'..., over proper 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 $\gamma \in \mathcal{T}_{Cfg}$, valuations $\rho : Var \to \mathcal{T}$ 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* $\rho' : Var \to \mathcal{T}$ *with* $\rho'(y) = \rho(y)$ *for all* $y \in Var \backslash X$

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

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

The pattern below matches the IMP configurations holding the code that sums the numbers up to `n` and a state binding program variables `s`,`n` to integers s,n, and $n \geq_{Int} 0$. We use `typewriter` fonts for program variables and *italic* for mathematical variables.

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

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

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

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

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

With the notation in Def. 4, $(\gamma, \rho) \models \varphi$ iff $\rho^{\gamma} \models_{\text{FOL}} \varphi^{\Box}$, and $\models \varphi$ iff $\mathcal{T} \models_{\text{FOL}} \varphi^{\Box}$. 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 Reduction

In [18] we showed that matching logic patterns can be used to give Hoare-equivalent but forwards axiomatic semantics and without introducing new quantifiers. Unfortunately, that approach shares a major disadvantage with other axiomatic approaches: the target language needs to be given a new, axiomatic semantics. Axiomatic semantics are less intuitive than operational semantics, are not easily executable, and are hard to test and prove sound. What we want is *one* formal semantics of a programming language, which should be both executable and suitable for program verification. In this section we introduce the novel concept of *matching logic reduction rule*, and we show that it captures operational semantics and can be used to specify properties about programs. Assume some arbitrary but fixed configuration signature Σ and model $\mathcal{T}$, like in Sections 2 and 3.

Definition 5. *A (matching logic) **reduction 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) **reduction system** is a recursively enumerable set of reduction rules. A reduction system $\mathcal{S}$ induces a **transition system** $(\mathcal{T}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$ on the configuration model $\mathcal{T}$: $\gamma \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma'$ 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 $(\gamma', \rho) \models \varphi'$. Reduction 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 (i.e., a predicate over the variables in l and r, containing no reductions). These conventional reduction rules are just special matching logic reduction rules. Indeed, a rule "$l \Rightarrow r$ if b" can be seen as syntactic sugar for the matching logic reduction rule $l \wedge b \Rightarrow r$: they specify the same transitions $\gamma \Rightarrow_S^{\mathcal{T}} \gamma'$ 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 well-defined (pick γ' to be $\rho(r)$).

Therefore, any language/calculus/system operational semantics defined using reduction rules corresponding to any of the styles enumerated in Section 2 *is* a particular matching logic reduction system. But how expressive are matching logic's reduction rules? Can they express more than just one-step transitions? Can they express meaningful program properties? The answer is *yes*, but we have to relax the meaning of $\varphi \Rightarrow \varphi'$ from "one step" to "zero, one or more steps". Consider 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 $n \geq_{Int} 0$, 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*.

In fact, we can associate a matching logic reduction rule to *any* IMP Hoare triple $\{\psi\}\ \texttt{code}\ \{\psi'\}$: $\exists X_{\texttt{code}}(\langle \texttt{code}, \sigma_{X_{\texttt{code}}}\rangle \wedge \psi_X) \Rightarrow \exists X_{\texttt{code}}(\langle \texttt{skip}, \sigma_{X_{\texttt{code}}}\rangle \wedge \psi'_X)$, 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 $\{\texttt{n = oldn} \wedge \texttt{n} \geq 0\}\ \texttt{SUM}\ \{\texttt{s = oldn*(oldn+1)/2} \wedge \texttt{n} = 0\}$. 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 reduction rule is

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

While this reduction 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 matching logic reduction rules as an alternative to Hoare triples for specifying program properties. Note, however, that matching logic reduction rules are strictly more expressive than Hoare triples, because

they can allow any code in their RHS patterns, not only `skip`. In fact, replacing Hoare logic reasoning by matching logic reduction reasoning using mechanical translations like above is discouraged in practice, because one would be required to still provide a Hoare logic for the target language, like in the current state-of-the-art [1, 17]. A strong point of our approach is that one does *not* have to go through this tedious step. We have implemented the proof system in Section 5 and verified dozens of challenging programs (see Section 6) with it applying only the operational semantic rules of the language and without having to prove any Hoare logic proof rules as lemmas. The reason we showed this translation was only to argue that the matching logic reduction rules are expressive.

We next define semantic validity in matching logic reduction. In conventional axiomatic semantics, a (partial correctness) Hoare triple is semantically valid, written $\models \{\psi\}\,\texttt{code}\,\{\psi'\}$, iff for any state $s \models \psi$, if `code` executed in state s terminates with state s' then $s' \models \psi'$. This elegant definition has the luxury of relying on another semantics of the language, which provides the notions of "execution", "termination", and "state". Since here all these happen in the same semantics given as a reduction system, and since the closest matching logic element to a "state" is a ground configuration in $\mathcal{T}_{Cfg}$ including both the state and the code, and since the transition system $(\mathcal{T}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$ gives all the operational behaviors of the defined language, we introduce the following

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

As already mentioned, φ' needs not have an empty (i.e., `skip` in the case of IMP) code cell. If φ' has an empty code cell 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 any of the two matching logic reduction rules $\varphi \Rightarrow \varphi'$ for `SUM` in this section. Unfortunately, such proofs are tedious, involving low-level details about the IMP transition system and induction. What we want is an abstract proof system for deriving matching logic reduction rules, which does not refer to the low-level transition system.

5 Proof System and Partial Correctness

We have seen that matching logic reduction rules can express both operational semantics rules and program specifications. Can we then build a unified theory of operational and axiomatic semantics, with the desirable features discussed in Section 1, based on the notion of a matching logic reduction rule? We next propose a language-independent proof system that allows us to start with a set of reduction 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 reduction rules as axioms. In particular, no auxiliary lemmas corresponding to Hoare logic rules are proved or needed.

Fig. 3 shows our nine-rule proof system for deriving matching logic reduction rules. Initially, $\mathcal{A}$ contains the operational semantics of the target language. The first group of rules (Reflexivity, Axiom, Substitution, Transitivity) have an operational nature and are

Rules of operational nature

Reflexivity :

$$\frac{}{\mathcal{A} \vdash \varphi \Rightarrow \varphi}$$

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 proof system

needed to execute reduction systems; any executable semantic framework is expected to have similar rules (see, e.g., rewriting logic [14]). The second group of rules (Case Analysis, Logic Framing, Consequence and Abstraction) have a deductive nature and are inspired from the subset of language-independent rules of Hoare logic [10].

The Circularity proof rule is new. It *language-independently* captures the various circular behaviors that appear in languages, due to loops, recursion, jumps, etc.

Definition 7. *Let $\mathcal{A} \vdash \varphi \Rightarrow^+ \varphi'$ be the derivation relation obtained by dropping the Reflexivity rule from the proof system in Fig. 3.*

The intuition for $\mathcal{A} \vdash \varphi \Rightarrow^+ \varphi'$ is that a configuration satisfying φ needs at least one operational semantics step to transit 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 reduction 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* [19]. 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. ***(partial correctness)*** *Let $\mathcal{S}$ be a well-defined and deterministic matching logic reduction 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 $\mathcal{S} \models \varphi \Rightarrow \varphi'$ (see Appendix A for the proof).*

Hence, proof derivations with the system in Fig. 3 are sound w.r.t. the (transition system generated by the) operational semantics, in the sense of partial correctness. The well-definedness requirement is acceptable (operational semantics satisfy it) and needed (otherwise not even the axioms satisfy $\mathcal{S} \models \varphi \Rightarrow \varphi'$). The determinism requirement is *unnecessary* in the proof of Theorem 1, but it makes the result mean partial correctness. Without it, $\mathcal{S} \models \varphi \Rightarrow \varphi'$ (see Definition 6) means "any terminating configuration that matches φ reduces, on some possible execution path, to a configuration that matches φ'". Determinism enforces "some path" to be equal to "all paths".

We illustrate our proof system by means of some 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 next property given as a matching logic reduction 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 $\varphi_{\text{LHS}} \Rightarrow \langle \texttt{skip}; \texttt{LOOP},\ (\texttt{s} \mapsto 0,\ \texttt{n} \mapsto n) \rangle \ \wedge\ n \geq_{Int} 0$. 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 reduction rule for `SUM` in Section 4, say $\mu^2_{\texttt{SUM}}$. Instead, let us prove the stronger result that the two matching logic reduction 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 reduction 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.

Following an approach similar to that in [18], we can show that, in the case of IMP, any property derived using its 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 reduction 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. Thus, one can argue that for the particular case of the IMP language, our proof system in Fig. 3 is relatively complete. We believe that one can prove a generic relative completeness result for matching logic reduction, but that is beyond the scope of this paper.

6 Implementation in MatchC

The main 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 needs to be provided in specifications. To test the practical effectiveness of matching logic reduction, we picked a fragment of C, called KernelC, and implemented a proof-of-concept program verifier for it based on matching logic, named MatchC. KernelC is quite expressive, including functions, structures, pointers and I/O primitives. MatchC uses matching logic reduction rules for program specifications and its implementation is directly based on the proof system in Fig. 3. It uses the operational semantics of KernelC *completely unchanged* for program verification.

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. In all these, the users only provide the program specifications, as matching logic reduction rules but using a user-friendly annotation-based interface, in addition to the unavoidable formalizations of the mathematical domains of interest. 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 reduction system giving the semantics of KernelC, and let C be the set of reduction 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 reduces φ iteratively using the rules in $\mathcal{S} \cup C$ searching for a formula that implies φ'. An SMT solver (Z3 [16]) is invoked to solve the side conditions of the rules. Whenever the reduction 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 [18], was discussed in [21]. 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. Its practical benefits may be considerable. We now only define *one* semantics of the target language, which is operational and thus well-understood and comparatively easier than defining an axiomatic semantics. Moreover, the semantics 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. This has already been done for C [5]. Then, we take this semantics and use it *as is* for program verification. Not only that we now completely skip 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 [7]. However, unlike ours, Floyd's rules are language-specific, not executable, and introduce quantifiers. Dynamic logic [2,9] extends FOL with modal operators to embed program fragments within program specifications. Like in matching logic, programs and specifications coexits in the same logic. However, unlike in our approach, one still needs to define an alternative (dynamic logic) semantics of the language, with language-specific proof rules, as one cannot use a conventional operational semantics with a language-independent proof system.

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. The relationship to Hoare logic was only sketched; details need to be worked out. 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 and formal proof objects produced. Ultimately, we would like to have a generic verifier taking an operational semantics as input, together with extension allowing users to provide pattern annotations, and to yield an automated program verifier based on the proof system in Fig. 3.

References

1. Appel, A.W.: Verified software toolchain. In: ESOP. LNCS, vol. 6602, pp. 1–17 (2011)

2. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software: The KeY Approach, LNCS, vol. 4334. Springer (2007)
3. Berry, G., Boudol, G.: The chemical abstract machine. Th. Comp. Sci. 96(1), 217–248 (1992)
4. Clavel, M., Durán, F., Eker, S., Meseguer, J., Lincoln, P., Martí-Oliet, N., Talcott, C.: All About Maude, LNCS, vol. 4350 (2007)
5. Ellison, C., Roşu, G.: An executable formal semantics of C with applications. In: POPL. pp. 533–544 (2012)
6. Felleisen, M., Findler, R.B., Flatt, M.: Semantics Engineering with PLT Redex. MIT (2009)
7. Floyd, R.W.: Assigning meaning to programs. In: Symposium on Applied Mathematics. vol. 19, pp. 19–32 (1967)
8. George, C., Haxthausen, A.E., Hughes, S., Milne, R., Prehn, S., Pedersen, J.S.: The RAISE Development Method. BCS Practitioner Series, Prentice Hall (1995)
9. Harel, D., Kozen, D., Tiuryn, J.: Dynamic logic. In: Handbook of Philosophical Logic. pp. 497–604 (1984)
10. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM 12(10), 576–580 (1969)
11. Hoare, C.A.R., Jifeng, H.: Unifying Theories of Programming. Prentice Hall (1998)
12. Jacobs, B.: Weakest pre-condition reasoning for java programs with JML annotations. J. Log. Algebr. Program. 58(1-2), 61–88 (2004)
13. Liu, H., Moore, J.S.: Java program verification via a JVM deep embedding in ACL2. In: TPHOLs. LNCS, vol. 3223, pp. 184–200 (2004)
14. Meseguer, J.: Conditioned rewriting logic as a united model of concurrency. Theor. Comput. Sci. 96(1), 73–155 (1992)
15. Mosses, P.D.: CASL Reference Manual, LNCS, vol. 2960. Springer (2004)
16. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS'08. LNCS, vol. 4963
17. Nipkow, T.: Winskel is (almost) right: Towards a mechanized semantics textbook. Formal Aspects of Computing 10, 171–186 (1998)
18. Rosu, G., Ellison, C., Schulte, W.: Matching logic: An alternative to Hoare/Floyd logic. In: AMAST. LNCS, vol. 6486, pp. 142–162 (2010)
19. Rosu, G., Lucanu, D.: Circular coinduction: A proof theoretical foundation. In: CALCO. LNCS, vol. 5728, pp. 127–144 (2009)
20. Rosu, G., Serbanuta, T.F.: An overview of the K semantic framework. J. Log. Algebr. Program. 79(6), 397–434 (2010)
21. Rosu, G., Stefanescu, A.: Matching logic: a new program verification approach (NIER track). In: ICSE. pp. 868–871 (2011)
22. 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)

A Proof of Theorem 1

Theorem 1. ***(partial correctness)*** *Let $\mathcal{S}$ be a well-defined and deterministic matching logic reduction system (corresponding to an operational semantics), and let $\mathcal{S} \vdash \varphi_{LHS} \Rightarrow \varphi_{RHS}$ be a sequent derived with the proof system in Fig. 3. Then $\mathcal{S} \models \varphi_{LHS} \Rightarrow \varphi_{RHS}$.*

Proof. Recall that $\mathcal{T}$ is a fixed configuration model, and $\mathcal{S}$ a fixed recursively enumerable, well-defined and deterministic set of reduction rules. We define the relation $(\mathcal{T}, <^{\mathcal{S}})$ as follows:

$$\gamma_1 <^{\mathcal{S}} \gamma_2 \text{ iff } \gamma_1 \text{ and } \gamma_2 \text{ terminate in } (\mathcal{T}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}}) \text{ and } \gamma_2 \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma_1$$

We notice that $<^{\mathcal{S}}$ has the following properties:

irreflexivity if $\gamma \in \mathcal{T}$ terminates in $(\mathcal{T}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$, then $\neg(\gamma \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma)$, hence $\neg(\gamma <^{\mathcal{S}} \gamma)$;

asymmetry if $\gamma_1, \gamma_2 \in \mathcal{T}$ terminate in $(\mathcal{T} \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$, and $\gamma_1 <^{\mathcal{S}} \gamma_2$, then $\gamma_2 \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma_1$ and $\neg(\gamma_1 \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma_2)$, hence $\neg(\gamma_2 <^{\mathcal{S}} \gamma_1)$;

transitivity if $\gamma_1, \gamma_2, \gamma_3 \in \mathcal{T}$ terminate in $(\mathcal{T}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$, and $\gamma_1 <^{\mathcal{S}} \gamma_2$ and $\gamma_2 <^{\mathcal{S}} \gamma_3$, then $\gamma_2 \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma_1$ and $\gamma_3 \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma_2$; by the transitivity of $\Rightarrow_{\mathcal{S}}^{+\mathcal{T}}$, it follows that $\gamma_3 \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma_1$, hence $\gamma_1 <^{\mathcal{S}} \gamma_3$.

We can conclude that $<^{\mathcal{S}}$ is a partial order relation. Moreover, since any decreasing chain has associated a reduction sequence containing only terms which terminate in $(\mathcal{T}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$, it follows that there are no infinite decreasing chains, or equivalently, that $<^{\mathcal{S}}$ is well-founded.

Let $\mathcal{P}_{fixed}$ be a fixed matching logic proof tree deriving the sequent $\mathcal{S} \vdash \varphi_{LHS} \Rightarrow \varphi_{RHS}$. We consider the domain $\mathcal{D}$ of triples

$$(\gamma, \mathcal{A}, \mathcal{P})$$

where $\gamma \in \mathcal{T}$ is a ground configuration that terminates in $(\mathcal{T}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$, $\mathcal{A}$ is a recursively enumerable set of reduction rules appearing in one of the sequents in $\mathcal{P}_{fixed}$, and $\mathcal{P}$ is a matching logic proof subtree of $\mathcal{P}_{fixed}$. Notice that since $\mathcal{P}_{fixed}$ is fixed, both $\mathcal{A}$ and $\mathcal{P}$ range over a finite number of instances. We define the lexicographical order $(\mathcal{D}, \prec)$ as follows:

$$\begin{aligned}
&(\gamma_1, \mathcal{A}_1, \mathcal{P}_1) \prec (\gamma_2, \mathcal{A}_2, \mathcal{P}_2)\\
&\text{iff } \gamma_1 <^{\mathcal{S}} \gamma_2\\
&\text{or } \gamma_1 = \gamma_2 \text{ and } \mathcal{A}_1 \subsetneq \mathcal{A}_2\\
&\text{or } \gamma_1 = \gamma_2 \text{ and } \mathcal{A}_1 = \mathcal{A}_2 \text{ and } \mathcal{P}_1 \text{ is a proper subtree of } \mathcal{P}_2
\end{aligned}$$

We notice that $\prec$ is a lexicographic order based on three well-founded partial order relations, namely: (1) $(\mathcal{T}, <^{\mathcal{S}})$; (2) strict inclusion relation ($\subsetneq$) on a finite number of sets; and (3) proper subtree relation on proof trees. It is known that the lexicographic ordering of sequences of fixed length [1] based on well-founded orders for each component of the sequence is itself well-founded, thus $\prec$ is well-founded.

Definition 8. *If $(\gamma, \mathcal{A}, \mathcal{P}) \in \mathcal{D}$ is such that $\mathcal{P}$ derives the sequent $\mathcal{A} \vdash \varphi \Rightarrow \varphi'$, then let Prop$(\gamma, \mathcal{A}, \mathcal{P})$ be the following property: for all $\rho : Var \to \mathcal{T}$ such that $(\gamma, \rho) \models \varphi$,*

[1] Note that this is not true for sequences or arbitrary lengths. For example, the set of finite sequences of elements of $(\{0, 1\}, <)$ is not well-founded, even if $<$ is well-founded on $\{0, 1\}$.

there exists a γ' such that $\gamma \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma'$ and $(\gamma', \rho) \models \varphi$; moreover, if $\mathcal{P}$ does not use the **Reflexivity** *rule, or the last step in $\mathcal{P}$ is a* **Circularity** *step, then $\gamma \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma'$ (that is, γ reduces to γ' in one or more steps).*

With these definitions, it is easy to see that Theorem 1 becomes a corollary of the following result: for all $\gamma \in \mathcal{T}$ such that γ terminates in $(\mathcal{T}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$, the property $Prop(\gamma, \mathcal{S}, \mathcal{P}_{fixed})$ holds. Also, note that the triple $(\gamma, \mathcal{S}, \mathcal{P}_{fixed})$ belongs to $\mathcal{D}$ for all the terminating γ. These observations imply that it suffices to prove the following more general result:

Lemma 1. *$Prop(\gamma, \mathcal{A}, \mathcal{P})$ holds for all $(\gamma, \mathcal{A}, \mathcal{P}) \in \mathcal{D}$.*

We prove Lemma 1 by well-founded induction on the $\prec$ partial order. Let us pick a $(\gamma, \mathcal{A}, \mathcal{P}) \in \mathcal{D}$ and let $\rho : Var \to \mathcal{T}$ be such that (γ, ρ) satisfy the left-hand-side formula of the rule derived by $\mathcal{P}$. We have the following cases based on the structure of $\mathcal{P}$:

- $\mathcal{P}$ is a **Reflexivity** step:

$$\frac{}{\mathcal{A} \vdash \varphi \Rightarrow \varphi}$$

  We pick γ' to be γ. Trivially, we have that $\gamma \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma'$, and $(\gamma', \rho) \models \varphi$, so we are done.
- $\mathcal{P}$ is an **Axiom** step:

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

  We distinguish two cases:
  - $\varphi \Rightarrow \varphi'$ belongs to $\mathcal{S}$. Since $\mathcal{S}$ is well-defined, there exists a γ' such that $(\gamma', \rho) \models \varphi'$. By the definition of the transition system $(\mathcal{T}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$, we have that $\gamma \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma'$, which implies that $\gamma \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma'$, and we are done.
  - $\varphi \Rightarrow \varphi'$ belongs to $\mathcal{A} \setminus \mathcal{S}$. Since the only way to add rules to the set of axioms is by using the **Circularity** rule, there must be a set of axioms $\mathcal{A}'$ and a proof tree $\mathcal{P}'$ deriving the sequent $\mathcal{A}' \vdash \varphi \Rightarrow \varphi'$, such that $\mathcal{P}'$ is a subtree of $\mathcal{P}_{fixed}$, and $\mathcal{P}$ is a leaf of $\mathcal{P}'$. Rules cannot be dropped from the set of axioms as we traverse a matching logic proof tree bottom-up, hence it must be the case that $\mathcal{A}' \subsetneq \mathcal{A}$. It follows that $(\gamma, \mathcal{A}', \mathcal{P}') \prec (\gamma, \mathcal{A}, \mathcal{P})$. By the induction hypothesis, $Prop(\gamma, \mathcal{A}', \mathcal{P}')$ holds. The last step in $\mathcal{P}'$ is a **Circularity** step, hence there exists a γ' such that $\gamma \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma'$ and $(\gamma', \rho) \models \varphi'$, so we are done.
- The last step in $\mathcal{P}$ is a **Substitution** step:

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

  Let $\rho' = \theta(\rho)$. It follows that $(\gamma, \rho') \models \varphi$. By the induction hypothesis there exists a γ' such that $\gamma \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma'$ and $(\gamma', \rho') \models \varphi'$. Then, we have that $(\gamma', \rho) \models \theta(\varphi')$. We notice that if $\mathcal{P}$ does not use the **Reflexivity** rule, then $\gamma \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma'$ by the induction hypothesis, and we are done.
- The last step in $\mathcal{P}$ is a **Transitivity** step:

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

Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be the proof trees deriving $\mathcal{A} \vdash \varphi_1 \Rightarrow \varphi_2$ and $\mathcal{A} \vdash \varphi_2 \Rightarrow \varphi_3$. Notice that $\mathcal{P}_1$ is a subtree of $\mathcal{P}$, hence $(\gamma, \mathcal{A}, \mathcal{P}_1) \prec (\gamma, \mathcal{A}, \mathcal{P})$. By the induction hypothesis, there exists a γ'' such that $\gamma \Rightarrow_S^{\star\mathcal{T}} \gamma''$ and $(\gamma'', \rho) \models \varphi_2$. Since $\mathcal{P}_2$ is also a subtree of $\mathcal{P}$ and either $\gamma'' <^S \gamma$ or $\gamma'' = \gamma$, it follows that $(\gamma'', \mathcal{A}, \mathcal{P}_2) \prec (\gamma, \mathcal{A}, \mathcal{P})$. Also by the induction hypothesis, there exists a γ' such that $\gamma'' \Rightarrow_S^{\star\mathcal{T}} \gamma'$ and $(\gamma', \rho) \models \varphi_3$. Due to the transitivity of the transition relation $\Rightarrow_S^{\star\mathcal{T}}$ in $\mathcal{T}$, it follows that $\gamma \Rightarrow_S^{\star\mathcal{T}} \gamma'$. We notice that if $\mathcal{P}$ does not use the **Reflexivity** rule, then $\gamma \Rightarrow_S^{+\mathcal{T}} \gamma''$ and $\gamma'' \Rightarrow_S^{+\mathcal{T}} \gamma'$ by the induction hypothesis. This implies that $\gamma \Rightarrow_S^{+\mathcal{T}} \gamma'$, and we are done.

– The last step in $\mathcal{P}$ is a **Case analysis** step:

$$\frac{\mathcal{A} \vdash \varphi_1 \Rightarrow \varphi \quad \mathcal{A} \vdash \varphi_2 \Rightarrow \varphi}{\mathcal{A} \vdash \varphi_1 \vee \varphi_2 \Rightarrow \varphi}$$

By the definition of satisfaction for disjunction, $(\gamma, \rho) \models \varphi_1 \vee \varphi_2$ implies $(\gamma, \rho) \models \varphi_1$ or $(\gamma, \rho) \models \varphi_2$. We can assume without loss of generality that $(\gamma, \rho) \models \varphi_1$. By the induction hypothesis there exists a γ' such that $\gamma \Rightarrow_S^{\star\mathcal{T}} \gamma'$ and $(\gamma', \rho) \models \varphi$. We notice that if $\mathcal{P}$ does not use the **Reflexivity** rule, then $\gamma \Rightarrow_S^{+\mathcal{T}} \gamma'$ by the induction hypothesis, and we are done.

– The last step in $\mathcal{P}$ is a **Logic framing** step:

$$\frac{\mathcal{A} \vdash \varphi \Rightarrow \varphi' \quad \psi \text{ is a FOL}_= \text{ formula}}{\mathcal{A} \vdash \varphi \wedge \psi \Rightarrow \varphi' \wedge \psi}$$

By the definition of satisfaction for conjunction and FOL formulae, $(\gamma, \rho) \models \varphi \wedge \psi$ implies that $(\gamma, \rho) \models \varphi$ and $\rho \models \psi$. By the induction hypothesis, there exists a γ' such that $\gamma \Rightarrow_S^{\star\mathcal{T}} \gamma'$ and $(\gamma', \rho) \models \varphi'$. It follows that $(\gamma', \rho) \models \varphi' \wedge \psi$. We notice that if $\mathcal{P}$ does not use the **Reflexivity** rule, then $\gamma \Rightarrow_S^{+\mathcal{T}} \gamma'$ by the induction hypothesis, and we are done.

– The last step in $\mathcal{P}$ is a **Consequence** step:

$$\frac{\models \varphi_1 \rightarrow \varphi_1' \quad \mathcal{A} \vdash \varphi_1' \Rightarrow \varphi_2' \quad \models \varphi_2' \rightarrow \varphi_2}{\mathcal{A} \vdash \varphi_1 \Rightarrow \varphi_2}$$

Since $\models \varphi_1 \rightarrow \varphi_1'$ and $(\gamma, \rho) \models \varphi_1$, it follows that $(\gamma, \rho) \models \varphi_1'$. By the induction hypothesis, there exists a γ' such that $\gamma \Rightarrow_S^{\star\mathcal{T}} \gamma'$ and $(\gamma', \rho) \models \varphi_2'$. Since $\models \varphi_2' \rightarrow \varphi_2$, it follows that $(\gamma', \rho) \models \varphi_2$. We notice that if $\mathcal{P}$ does not use the **Reflexivity** rule, then $\gamma \Rightarrow_S^{+\mathcal{T}} \gamma'$by the induction hypothesis, and we are done.

– The last step in $\mathcal{P}$ is an **Abstraction** step:

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

Since the variables in X do not appear free in φ', and $(\gamma, \rho) \models \exists X \varphi$, by the definition of the satisfaction for existential quantification, there exists a substitution ρ' that agrees with ρ on the free variables of φ', such that $(\gamma, \rho') \models \varphi$. By the induction hypothesis, there exists γ' such that $\gamma \Rightarrow_S^{\star\mathcal{T}} \gamma'$ and $(\gamma', \rho') \models \varphi'$. Since ρ' and ρ agree on the free variables of φ', we can conclude that $(\gamma', \rho) \models \varphi'$. We notice that if $\mathcal{P}$ does not use the **Reflexivity** rule, then $\gamma \Rightarrow_S^{+\mathcal{T}} \gamma'$ by the induction hypothesis, and we are done.

– The last step in $\mathcal{P}$ is a **Circularity** step:

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

Let $\mathcal{P}^{+}$ and $\mathcal{P}'$ be the proof trees for the sequents $\mathcal{A} \vdash \varphi \Rightarrow^{+} \varphi''$ and $\mathcal{A} \cup \{\varphi \Rightarrow \varphi'\} \vdash \varphi'' \Rightarrow \varphi'$. Notice that $\mathcal{P}^{+}$ is a subtree of $\mathcal{P}$, hence $(\gamma, \mathcal{A}, \mathcal{P}^{+}) \prec (\gamma, \mathcal{A}, \mathcal{P})$. By the induction hypothesis, $Prop(\gamma, \mathcal{A}, \mathcal{P}^{+})$ holds. Since $\mathcal{P}^{+}$ does not use the **Reflexivity** rule, there exists a γ'' such that $\gamma \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma''$ and $(\gamma'', \rho) \models \varphi''$. It follows that $\gamma'' <^{\mathcal{S}} \gamma$, so $(\gamma'', \mathcal{A} \cup \{\varphi'' \Rightarrow \varphi'\}, \mathcal{P}') \prec (\gamma, \mathcal{A}, \mathcal{P})$. Then $Prop(\gamma'', \mathcal{A} \cup \{\varphi'' \Rightarrow \varphi'\}, \mathcal{P}')$ holds by the induction hypothesis. Hence, there exists γ' such that $\gamma'' \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma'$ and $(\gamma', \rho) \models \varphi'$. Due to the transitivity of the transition relation $\Rightarrow_{\mathcal{S}}^{\star\mathcal{T}}$ in $\mathcal{T}$, it follows that $\gamma \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma''$, and we are done.