FM'12, LNCS 7436, pp 387-402. 2012

From Hoare Logic to Matching Logic Reachability ⋆

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

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

Abstract. Matching logic reachability has been recently proposed as an alternative program verification approach. Unlike Hoare logic, where one defines a language-specific proof system that needs to be proved sound for each language separately, matching logic reachability provides a *language-independent* and *sound* proof system that directly uses the trusted operational semantics of the language as axioms. Matching logic reachability thus has a clear practical advantage: it eliminates the need for an additional semantics of the same language in order to reason about programs, and implicitly eliminates the need for tedious soundness proofs. What is not clear, however, is whether matching logic reachability is as powerful as Hoare logic. This paper introduces a technique to mechanically translate Hoare logic proof derivations into equivalent matching logic reachability proof derivations. The presented technique has two consequences: first, it suggests that matching logic reachability has no theoretical limitation over Hoare logic; and second, it provides a new approach to prove Hoare logics sound.

1 Introduction

Operational semantics are undoubtedly one of the most accessible semantic approaches. Language designers typically do not need an extensive theoretical background in order to define an operational semantics to a language, because they can think of it as if "implementing" an interpreter for the language. For example, consider the following two rules from the (operational) reduction semantics of a simple imperative language:

$$\texttt{while}(e)\ s \ \Rightarrow\ \texttt{if}(e)\ s;\ \texttt{while}(e)\ s\ \texttt{else skip}$$
$$\texttt{proc}() \ \Rightarrow\ \texttt{body} \qquad \text{where "}\texttt{proc}()\,\texttt{body}\text{" is a procedure}$$

The former says that loops are unrolled and the second says that procedure calls are inlined (for simplicity, we assumed no-argument procedures and no local variables). In addition to accessibility, operational semantics have another major advantage: they can be efficiently executable, and thus testable. For example, one can test an operational semantics as if it was an interpreter or a compiler, by executing large test suites of programs. This way, semantic or design flaws can be detected and confidence in the semantics can be incrementally build. We refer the interested reader to [1, 3, 6] for examples of large operational semantics (for C) and examples of how they are tested. Because of all the above, it is quite common that operational semantics are considered trusted reference models of the programming languages they define, and thus serve as a formal basis for language understanding, design, and implementation.

⋆ Full version of this paper, with proofs, available at http://hdl.handle.net/2142/31335.

With few notable exceptions, e.g. [10], operational semantics are typically considered inappropriate for program verification. That is to a large extent due to the fact that program reasoning with an operational semantics typically reduces to reasoning within the transition system associated to the operational semantics, which can be quite low level. Instead, semantics which are more appropriate for program reasoning are typically given to programming languages, such as axiomatic semantics in the form of Hoare logic proof systems for deriving Hoare triples $\{\mathit{precondition}\}\ \texttt{code}\ \{\mathit{postcondition}\}$. For example, the proof rules below correspond to the operational semantics rules above:

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

$$\frac{\mathcal{H} \cup \{\psi\}\,\texttt{proc()}\,\{\psi'\} \vdash \{\psi\}\,\texttt{body}\,\{\psi'\}}{\mathcal{H} \vdash \{\psi\}\,\texttt{proc()}\,\{\psi'\}} \quad \text{where "}\texttt{proc() body}\text{" is a procedure}$$

The second rule takes into account the fact that the procedure `proc` might be recursive (the claimed procedure specification is assumed as hypothesis when deriving its body's property). One may need to use several instances of this rule in order to derive the properties of mutually recursive procedures. Both proof rules above define the notion of an invariant, the former for while loops (we assume a C-like language, where zero means false and non-zero means true) and the latter for recursive procedures. These proof rules are so compact only because we are making (unrealistic) simplifying assumptions about the language. Hoare logic proof systems for real languages are quite involved (see, e.g., [1] for C and [9] for Java), which is why, for trusted verification, one needs to prove them sound with respect to more trusted (typically operational) semantics; the state-of-the-art approaches in mechanical verification do precisely that [1, 8–10, 12, 18].

Matching logic reachability [16] is a new program verification approach, based on operational semantics. Instead of proving properties at the low level of a transition system, matching logic reachability provides a high-level proof system for deriving program properties, like Hoare logic. State properties are specified as *(matching logic) patterns* [17], which are program configuration terms with variables, containing both program and state fragments like in operational semantics, but the variables can be constrained using logical formulae, like in Hoare logic. Program properties are specified as *reachability rules* $\varphi \Rightarrow \varphi'$ between patterns φ and φ'; intuitively, $\varphi \Rightarrow \varphi'$ states that a program configuration γ that *matches* pattern φ takes zero, one or more steps in the associated transition system to reach a configuration γ' that *matches* φ'. Unlike in Hoare logic, the matching logic reachability proof rules are all *language-independent*, taking as axioms an operational semantics given as a set of reachability rules. The key proof rule of matching logic reachability is Circularity, which language-independently captures the various circular behaviors in languages, due to loops, recursion, etc.

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

$\mathcal{A}$ initially contains the operational semantics rules. Circularity adds new reachability rules to $\mathcal{A}$ during the proof derivation process, which can be used in their own proof! Its correctness 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.

Everything else being equal, matching logic reachability has a clear pragmatic advantage over Hoare logic: it eliminates the need for an additional semantics of the same language, and implicitly eliminates the need for non-trivial and tedious correctness proofs. The soundness of matching logic reachability has already been shown in [16]. Its practicality and usability have been demonstrated by the MatchC automatic program verifier for a C fragment [14], which is a faithful implementation of the matching logic reachability proof system. What is missing is a formal treatment of its *completeness*. Since Hoare logic is relatively complete [5], any semantically valid program property expressed as a Hoare triple can also be derived using the Hoare logic proof system (provided an oracle that knows all the properties of the state model is available). Of course, since Hoare logic is language-specific, its relative completeness needs to be proved for each language individually. Nevertheless, such relative completeness proofs are quite similar and not difficult to adapt from one language to another.

This paper addresses the completeness of matching logic reachability. A technique to mechanically translate Hoare logic triples into reachability rules and Hoare logic proof derivations into equivalent matching logic reachability proof derivations is presented and proved correct. The generated matching logic reachability proof derivations are within a linear factor larger in size than the original Hoare logic proofs. Because of the language-specific nature of Hoare logic, we define and prove our translation in the context of a specific but canonical imperative language, IMP. However, the underlying idea is general. We also apply it to an extension with mutually recursive procedures.

Although we can now regard Hoare logic as a methodological fragment of matching logic reachability, where any Hoare logic proof derivation can be mimicked using the matching logic reachability proof system, experience with MatchC tells us that in general one should *not* want to verify programs following this route in practice. Specifying program properties and verifying them directly using the matching logic reachability capabilities, without going through its Hoare logic fragment, gives us shorter and more intuitive specifications and proofs. Therefore, in our view, the result of this paper should be understood through its theoretical value. First, it shows that matching logic reachability has no theoretical limitation over Hoare logic, in spite of being language-independent and working directly with the trusted operational semantics. Second, it provides a new and abstract way to prove Hoare logics sound, where one does not need to make use of low-level transition systems and induction, instead relying on the soundness of matching logic reachability (proved generically, for all languages).

The reminder of this paper is organized as follows. Section 2 recalls operational semantics and Hoare logic, by means of the IMP language. Section 3 recalls matching logic notions, including the sound proof system for matching logic reachability. Section 4 illustrates the differences between Hoare logic and matching logic reachability. Section 5 presents our translation technique and proves its correctness. Section 6 concludes.

2 IMP: Operational Semantics and Hoare Logic

Here we recall operational semantics, Hoare logic, and related notions, and introduce our notation and terminology for these. We do so by means of the simple IMP imperative language. Figure 1 shows its syntax, an operational semantics based on evaluation contexts [7], and a Hoare logic for it. IMP has only integer expressions, which can also

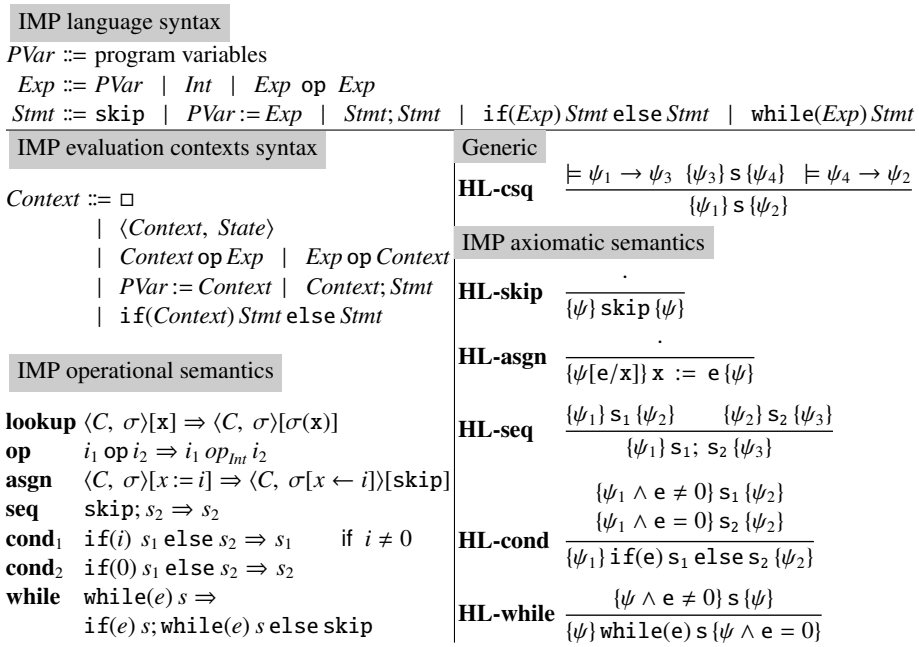

Fig. 1. IMP language syntax (top), operational semantics (left) and Hoare logic (right).

be used as conditions of `if` and `while` (zero means false and non-zero means true, like in C). Expressions are built with integer constants, program variables, and conventional arithmetic constructs. For simplicity, we only show a generic binary operation, `op`. IMP statements are the variable assignment, `if`, `while` and sequential composition.

The IMP 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 domains of integers (including arithmetic operations $i_1\ op_{Int}\ i_2$, etc.) and of maps (including lookup $\sigma(x)$ and update $\sigma[x \leftarrow i]$ operations). IMP's operational semantics has 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 and a redex, which form a configuration, and reduces the said configuration by rewriting the redex and possibly the context. As a notation, the context is skipped if not used. E.g., the rule **op** is in fact $\langle C,\ \sigma \rangle[i_1\ \texttt{op}\ i_2] \Rightarrow \langle C,\ \sigma \rangle[i_1\ op_{Int}\ i_2]$. The code context meta-variable C allows us to instantiate a schema into reduction rules, one for each redex of each code fragment. For example, $\langle \texttt{x := 5; y := 0},\ x \mapsto 0 \rangle$ can be split as $\langle \Box; \texttt{y := 0},\ x \mapsto 0 \rangle[\texttt{x := 5}]$, which by **asgn** reduces to $\langle \Box; \texttt{y := 0},\ x \mapsto 5 \rangle[\texttt{skip}]$, or equivalently to $\langle \texttt{skip; y := 0},\ x \mapsto 5 \rangle$.

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 boolean condition b. There are several operational semantics styles based on such rules. Besides the popular reduction semantics with evaluation contexts, we also have the chemical abstract machine [2] and $\mathbb{K}$ [13].

Large languages have been given semantics with only rules of the form "$l \Rightarrow r$ if b", including C [6] (defined in $\mathbb{K}$ with more than 1200 such rules). The matching logic reachability proof system works with any such rules (taking them as axioms), and is agnostic to the particular semantics or any other method used to produce them.

The major role of an operational semantics is to yield a canonical and typically trusted model of the defined language, as a transition system over program configurations. Such transition systems are important in this paper, so we formalize them here. We also recall some mathematical notions and notations, although we generally assume the reader is familiar with basic concepts of algebraic specification and first-order logic. Given an *algebraic signature* Σ, T_Σ denotes the *initial Σ-algebra* of ground terms (terms without variables), $T_\Sigma(Var)$ the *free Σ-algebra* of terms with variables in Var, and $T_{\Sigma,s}(Var)$ the set of Σ-terms of sort s. Valuations $\rho : Var \to \mathcal{T}$ with $\mathcal{T}$ a Σ-algebra extend uniquely to *morphisms of Σ-algebras* $\rho : T_\Sigma(Var) \to \mathcal{T}$. These notions extend to algebraic specifications. Many mathematical structures needed for language semantics have been defined as initial Σ-algebras: boolean algebras, natural/integer/rational numbers, lists, sets, bags (or multisets), maps (used as IMP's states), trees, queues, stacks, etc. We refer the reader to the CASL [11] and Maude [4] manuals for examples.

Let us fix the following: (1) an algebraic signature Σ, associated to some desired configuration syntax, with distinguished sorts *Cfg* and *Bool*; (2) a sort-wise infinite set of variables Var; and (3) a Σ-algebra $\mathcal{T}$, the *configuration model*, which may but need not necessarily be the initial or free Σ-algebra. As usual, $\mathcal{T}_{Cfg}$ denotes the elements of $\mathcal{T}$ of sort *Cfg*, which we call (concrete) *configurations*. Let $\mathcal{S}$ (for "semantics") be a set of reduction rules "$l \Rightarrow r$ if b" like above, where $l, r \in T_{\Sigma,Cfg}(Var)$ and $b \in T_{\Sigma,Bool}(Var)$.

Definition 1. *$\mathcal{S}$ yields a **transition system** $(\mathcal{T}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$ on the configuration model $\mathcal{T}$, where $\gamma \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma'$ for $\gamma, \gamma' \in \mathcal{T}_{Cfg}$ if and only if there exist a reduction rule "$l \Rightarrow r$ if b" in $\mathcal{S}$ and a valuation $\rho : Var \to \mathcal{T}$ such that $\rho(l) = \gamma$, $\rho(r) = \gamma'$ and $\rho(b)$ holds.*

$(\mathcal{T}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$ is a conventional transition system, i.e. a set with a binary relation on it (in fact, $\Rightarrow_{\mathcal{S}}^{\mathcal{T}} \subseteq \mathcal{T}_{Cfg} \times \mathcal{T}_{Cfg}$), and captures the operational behaviors of the language defined by $\mathcal{S}$.

Hence, an operational semantics defines a set of reduction rules which can be used in some implicit way to yield program behaviors. On the other hand, a Hoare logic defines a proof system that explicitly tells how to derive program properties formalized as Hoare triples. Operational semantics are easy to define, test and thus build confidence in, since we can execute them against benchmarks of programs; e.g., the C semantics have been extensively tested against compiler test-suites [3, 6]. On the other hand, Hoare logics are more involved and need to be proved sound w.r.t. another, more trusted semantics.

Definition 2. ***(partial correctness)** For the IMP language in Fig. 1, a Hoare triple $\{\psi\}\,\texttt{code}\,\{\psi'\}$ is **semantically valid**, written $\models \{\psi\}\,\texttt{code}\,\{\psi'\}$, if and only if for all states σ and σ', it is the case that if $\sigma \models \psi$ and $\langle \texttt{code}, \sigma\rangle$ terminates in $(\mathcal{T}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$ and $\langle \texttt{code}, \sigma\rangle \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \langle \texttt{skip}, \sigma'\rangle$ then $\sigma' \models \psi'$. The Hoare logic proof system in Fig. 1 is **sound** if and only if $\vdash \{\psi\}\,\texttt{code}\,\{\psi'\}$ implies $\models \{\psi\}\,\texttt{code}\,\{\psi'\}$.*

In Definition 2, we tacitly identified the ground configurations $\langle \texttt{code}, \sigma\rangle$ and $\langle \texttt{skip}, \sigma'\rangle$ with their (unique) interpretation in the configuration model $\mathcal{T}$. First-order logic (FOL) validity, both in Definition 2 and in the **HL-csq** in Fig. 1, is relative to $\mathcal{T}$. Partial correctness says the postcondition holds only when the program terminates. We do not address total correctness (i.e., the program must also terminate) in this paper.

3 Matching Logic Reachability

This section recalls matching logic and matching logic reachability notions and notations from [16, 17]. In matching logic reachability, *patterns* specify configurations and *reachability rules* specify operational transitions or program properties. A language-independent *proof system* takes a set of reachability rules (operational semantics) as axioms and derives new reachability rules (program properties). Matching logic is parametric in a model of program configurations. For example, as seen in Section 1, IMP's configurations are pairs $\langle \texttt{code}, \sigma \rangle$ with `code` a fragment of program and σ a *State*.

Like in Section 1, let us fix an algebraic signature Σ (of configurations) with a distinguished sort *Cfg*, a sort-wise infinite set of variables *Var*, and a (configuration) Σ-model $\mathcal{T}$ (which need not be the initial model T_Σ or the free model $T_\Sigma(Var)$).

Definition 3. *[17] A matching logic formula, or a **pattern**, is a first-order logic (FOL) formula which allows terms in $T_{\Sigma,Cfg}(Var)$, called **basic patterns**, as predicates. We define the satisfaction $(\gamma, \rho) \models \varphi$ 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* $\rho : Var \to \mathcal{T}$.

A basic pattern π is satisfied by all the configurations γ that *match* it; the ρ in $(\gamma, \rho) \models \pi$ can be thought of as the "witness" of the matching, and can be further constrained in a pattern. If `SUM` is the IMP code "`s:=0; while(n>0)(s:=s+n; n:=n-1)`", then $\exists s\ (\langle\ \texttt{SUM},\ (\texttt{s} \mapsto s,\ \texttt{n} \mapsto n)\ \rangle \wedge n \geq_{Int} 0)$ is a pattern matched by the configurations with code `SUM` and state mapping program variables `s`,`n` into integers s,n with n positive. Note that we use `typewriter` for program variables in *PVar* and *italic* for mathematical variables in *Var*. Pattern reasoning reduces to FOL reasoning in the model $\mathcal{T}$ [16].

Definition 4. *A (matching logic) **reachability rule** is a pair $\varphi \Rightarrow \varphi'$, where φ (the **left-hand side**, or **LHS**) and φ' (the **right-hand side**, or **RHS**), are matching logic patterns (with free variables). A **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: $\gamma \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma'$ for $\gamma, \gamma' \in \mathcal{T}_{Cfg}$ iff there exist $\varphi \Rightarrow \varphi'$ in $\mathcal{S}$ and $\rho : Var \to \mathcal{T}$ with $(\gamma, \rho) \models \varphi$ 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 exists $\gamma' \in \mathcal{T}_{Cfg}$ with $(\gamma', \rho) \models \varphi'$. 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.*

Operational semantics defined with rules "$l \Rightarrow r$ if b", like those in Section 2, are particular well-defined reachability systems with rules of the form $l \wedge b \Rightarrow r$. Intuitively, the first rule states that a ground configuration γ which is an instance of the term l and satisfies the boolean condition b reduces to an instance γ' of r. Matching logic was designed to express terms with constraints: $l \wedge b$ is satisfied by exactly all the γ above. Thus, matching logic reachability naturally captures reduction semantics (see [16] for more details). Reachability rules can also specify program properties. The rule

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

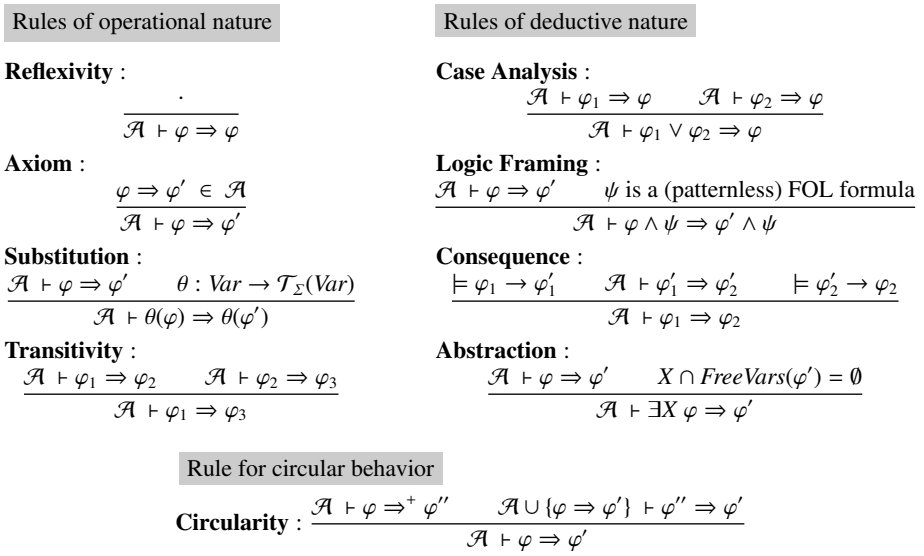

Fig. 2. Matching logic reachability proof system (nine language-independent proof rules).

specifies the property of `SUM`. Unlike Hoare triples, which only specify properties about the final states of programs, reachability rules can also specify properties about intermediate states (see the end of Section 4 for an example). Hoare triples correspond to particular rules with all the basic patterns in the RHS holding the code `skip`, like above.

Definition 5. *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 $\gamma \in \mathcal{T}_{Cfg}$ 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 $(\gamma', \rho) \models \varphi'$.*

Intuitively, $\mathcal{S} \models \varphi \Rightarrow \varphi'$ specifies reachability: any terminating configuration matching φ transits, on some execution path, to a configuration matching φ'. If $\mathcal{S}$ is deterministic, then "some path" is equivalent to "all paths", and thus $\varphi \Rightarrow \varphi'$ captures partial correctness. If φ' has the empty code `skip`, 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.

The above reachability rule for `SUM` is valid, although the proof is tedious, involving low-level IMP transition system details and induction. Figure 2 shows the *language-independent* matching logic reachability proof system which derives such rules while avoiding the transition system. Initially, $\mathcal{A}$ contains the operational semantics of the target language. Reflexivity, Axiom, Substitution, and Transitivity have an operational nature and derive concrete and (linear) symbolic executions. Case Analysis, Logic Framing, Consequence and Abstraction have a deductive nature. The Circularity proof rule has a coinductive nature and captures the various circular behaviors that appear in languages, due to loops, recursion, etc. Specifically, we can derive $\mathcal{A} \vdash \varphi \Rightarrow \varphi'$ whenever we can derive $\varphi \Rightarrow \varphi'$ by starting with one or more steps in $\mathcal{A}$ ($\Rightarrow^+$ means derivable without Reflexivity) and continuing with steps which can involve both rules from $\mathcal{A}$ and the rule to be proved itself, $\varphi \Rightarrow \varphi'$. For example, the first step can be a loop unrolling in the case of loops, or a function invocation in the case of recursive functions.

Theorem 1. ***(soundness)*** *[16] 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. 2. Then $\mathcal{S} \models \varphi \Rightarrow \varphi'$.*

4 Hoare Logic versus Matching Logic Reachability

This section prepares the reader for our main result, by illustrating the major differences between Hoare logic and matching logic reachability using examples. We show how the same program property can be specified both as a Hoare triple and as a matching logic reachability rule, and then how it can be derived using each of the two proof systems.

Recall the SUM program "`s:=0; while(n>0)(s:=s+n; n:=n-1)`" in IMP. Fig. 3 gives a Hoare logic proof that SUM adds the first natural numbers (bottom left column) and a matching logic reachability proof of the same property (bottom right column). The top contains some useful macros. For the explanations of these proofs below, "triple *n*" refers to the Hoare triple numbered with *n* in the bottom left column, and "rule *m*" refers to the matching logic sequent numbered with *m* in the bottom right column in Fig. 3.

The behavior of SUM can be specified by the Hoare triple $\{\psi_{\text{pre}}\}\,\texttt{SUM}\,\{\psi_{\text{post}}\}$, that is

$$\{\texttt{n} = \texttt{oldn} \wedge \texttt{n} \geq 0\}\,\texttt{SUM}\,\{\texttt{s} = \texttt{oldn*(oldn+1)/2} \wedge \texttt{n} = 0\}$$

The oldn variable is needed in order to remember the initial value of n. Let us derive this Hoare triple using the Hoare logic proof system in Fig. 1. We can derive our original Hoare triple by first deriving triples 1 and 5, namely

$$\{\texttt{n} = \texttt{oldn} \wedge \texttt{n} \geq 0\}\,\texttt{s:=0}\,\{\psi_{\text{inv}}\} \qquad \{\psi_{\text{inv}}\}\,\texttt{LOOP}\,\{\texttt{s} = \texttt{oldn*(oldn+1)/2} \wedge \texttt{n} = 0\}$$

and then using the proof rule **HL-seq** in Fig. 1. Triple 1 follows by **HL-asgn** and **HL-csq**. Triple 5 follows by **HL-while** from triple 4, which in turn follows from triples 2 and 3 by **HL-seq**. Finally, triples 2 and 3 follow each by **HL-asgn** and **HL-csq**.

Before we discuss the matching logic reachability proof derivation, let us recall some important Hoare logic facts. First, Hoare logic makes no theoretical distinction between program variables, which in the case of IMP are *PVar* constants, and mathematical variables, which in the case of IMP are variables of sort *Var*. For example, in the proof above, n as a program variable, n as an integer variable appearing in the state specifications, and oldn which appears only in state specifications but never in the program, were formally treated the same way. Second, the same applies to language arithmetic constructs versus mathematical domain operations. For example, there is no distinction between the + construct for IMP expressions and the $+_{Int}$ operation that the integer domain provides. Third, Hoare logic takes FOL substitution for granted (see **HL-asgn**), which in reality adds a linear complexity in the size of the FOL specification to the proof. These and other simplifying assumptions make proofs like above look simple and compact, but come at a price: expressions cannot have side effects. Since in many languages expressions do have side effects, programs typically suffer (possibly error-prone) transformations that extract and isolate the side effects into special statements. Also, in practice program verifiers do make a distinction between language constructs and mathematical ones, and appropriately translate the former into the latter in specifications.

Code macros:

$\texttt{SUM} \equiv \texttt{s:=0; while(n>0) (s:=s+n; n:=n-1)}$
$\texttt{LOOP} \equiv \texttt{while(n>0) (s:=s+n; n:=n-1)}$
$\texttt{BODY} \equiv \texttt{s:=s+n; n:=n-1}$
$\texttt{IF} \equiv \texttt{if(n>0) (s:=s+n; n:=n-1; while(n>0) (s:=s+n; n:=n-1)) else skip}$
$\texttt{S}_1 \equiv \texttt{s:=s+n; n:=n-1; while(n>0) (s:=s+n; n:=n-1)}$
$\texttt{S}_2 \equiv \texttt{n:=n-1; while(n>0) (s:=s+n; n:=n-1)}$

Hoare logic formula macros:

$\psi_{\text{pre}} \equiv \texttt{n} = \texttt{oldn} \wedge \texttt{n} \geq \texttt{0}$
$\psi_{\text{post}} \equiv \texttt{s} = \texttt{oldn*(oldn+1)/2} \wedge \texttt{n} = \texttt{0}$
$\psi_{\text{inv}} \equiv \texttt{s} = \texttt{(oldn-n)*(oldn+n+1)/2} \wedge \texttt{n} \geq \texttt{0}$
$\psi_1 \equiv \psi_{\text{inv}} \wedge \texttt{n} > \texttt{0}$
$\psi_2 \equiv \texttt{s} = \texttt{(oldn-n+1)*(oldn+n)/2} \wedge \texttt{n} > \texttt{0}$

Matching logic pattern macros:

$\varphi_{\text{LHS}} \equiv \langle \texttt{SUM}, (\texttt{s} \mapsto s, \texttt{n} \mapsto n) \rangle \wedge n \geq_{Int} 0$
$\varphi_{\text{RHS}} \equiv \langle \texttt{skip}, (\texttt{s} \mapsto n *_{Int} (n +_{Int} 1)/_{Int} 2, \texttt{n} \mapsto 0) \rangle$
$\varphi_{\text{inv}} \equiv \langle \texttt{LOOP}, (\texttt{s} \mapsto (n -_{Int} n') *_{Int} (n +_{Int} n' +_{Int} 1)/_{Int} 2, \texttt{n} \mapsto n') \rangle \wedge n' \geq_{Int} 0$
$\varphi_{\text{if}} \equiv \langle \texttt{IF}, (\texttt{s} \mapsto (n -_{Int} n') *_{Int} (n +_{Int} n' +_{Int} 1)/_{Int} 2, \texttt{n} \mapsto n') \rangle \wedge n' \geq_{Int} 0$
$\varphi_{\text{true}} \equiv \langle \texttt{IF}, (\texttt{s} \mapsto (n -_{Int} n') *_{Int} (n +_{Int} n' +_{Int} 1)/_{Int} 2, \texttt{n} \mapsto n') \rangle \wedge n' >_{Int} 0$
$\varphi_{\text{false}} \equiv \langle \texttt{IF}, (\texttt{s} \mapsto n *_{Int} (n +_{Int} 1)/_{Int} 2, \texttt{n} \mapsto 0) \rangle$
$\varphi_1 \equiv \langle \texttt{S}_1, (\texttt{s} \mapsto (n -_{Int} n') *_{Int} (n +_{Int} n' +_{Int} 1)/_{Int} 2, \texttt{n} \mapsto n') \rangle \wedge n' >_{Int} 0$
$\varphi_2 \equiv \langle \texttt{S}_2, (\texttt{s} \mapsto (n -_{Int} n' +_{Int} 1) *_{Int} (n +_{Int} n')/_{Int} 2, \texttt{n} \mapsto n') \rangle \wedge n' >_{Int} 0$
$\varphi_{\text{body}} \equiv \langle \texttt{LOOP}, (\texttt{s} \mapsto (n -_{Int} n' +_{Int} 1) *_{Int} (n +_{Int} n')/_{Int} 2, \texttt{n} \mapsto n' -_{Int} 1) \rangle \wedge n' >_{Int} 0$
$\mathcal{A}_{\text{LOOP}} \equiv \mathcal{S}_{\text{IMP}} \cup \{\varphi_{\text{inv}} \Rightarrow \varphi_{\text{RHS}}\}$

Hoare logic proof

Hoare triple	Proof rule	Adtl. Steps
1.$\{\psi_{\text{pre}}\}\ \texttt{s:=0}\ \{\psi_{\text{inv}}\}$	**HL-asgn**	1/17
2.$\{\psi_1\}\ \texttt{s:=s+n}\ \{\psi_2\}$	**HL-asgn**	1/17
3.$\{\psi_2\}\ \texttt{n:=n-1}\ \{\psi_{\text{inv}}\}$	**HL-asgn**	1/17
4.$\{\psi_1\}\ \texttt{BODY}\ \{\psi_{\text{inv}}\}$	**HL-seq**$(2,3)$	0/0
5.$\{\psi_{\text{inv}}\}\ \texttt{LOOP}\ \{\psi_{\text{post}}\}$	**HL-while**(4)	1/0
6.$\{\psi_{\text{pre}}\}\ \texttt{SUM}\ \{\psi_{\text{post}}\}$	**HL-seq**$(1,5)$	0/0

Matching logic reachability proof

Reachability	ASLF with	Steps
1.$\mathcal{S}_{\text{IMP}} \vdash \exists s\, \varphi_{\text{LHS}} \Rightarrow \exists n' \varphi_{\text{inv}}$	$\textbf{asgn}_{\texttt{s}}$, **seq**	1/0/1/1/0
2.$\mathcal{S}_{\text{IMP}} \vdash \varphi_{\text{inv}} \Rightarrow^{+} \varphi_{\text{if}}$	**while**	0/0/0/0/0
3.$\mathcal{A}_{\text{LOOP}} \vdash \varphi_{\text{true}} \Rightarrow \varphi_1$	$\textbf{lookup}_{\texttt{n}}$, $\textbf{op}_{>}$, $\textbf{cond}_1$	2/0/0/0/0
4.$\mathcal{A}_{\text{LOOP}} \vdash \varphi_1 \Rightarrow \varphi_2$	$\textbf{lookup}_{\texttt{n}}$, $\textbf{lookup}_{\texttt{s}}$, $\textbf{op}_{+}$, $\textbf{asgn}_{\texttt{n}}$, **seq**	4/0/0/0/0
5.$\mathcal{A}_{\text{LOOP}} \vdash \varphi_2 \Rightarrow \varphi_{\text{body}}$	$\textbf{lookup}_{\texttt{n}}$, $\textbf{op}_{-}$, $\textbf{asgn}_{\texttt{n}}$, **seq**	3/0/1/0/0
6.$\mathcal{A}_{\text{LOOP}} \vdash \varphi_{\text{body}} \Rightarrow \varphi_{\text{RHS}}$	$\varphi_{\text{inv}} \Rightarrow \varphi_{\text{RHS}}$	0/0/0/0/0
7.$\mathcal{A}_{\text{LOOP}} \vdash \varphi_{\text{false}} \Rightarrow \varphi_{\text{RHS}}$	$\textbf{lookup}_{\texttt{n}}$, $\textbf{op}_{>}$, $\textbf{cond}_2$	2/0/1/0/0
8.$\mathcal{A}_{\text{LOOP}} \vdash \varphi_{\text{if}} \Rightarrow \varphi_{\text{RHS}}$		3/1/1/0/0
9.$\mathcal{S}_{\text{IMP}} \vdash \varphi_{\text{inv}} \Rightarrow \varphi_{\text{RHS}}$		0/0/0/0/1
10.$\mathcal{S}_{\text{IMP}} \vdash \exists n' \varphi_{\text{inv}} \Rightarrow \varphi_{\text{RHS}}$		0/0/0/1/0
11.$\mathcal{S}_{\text{IMP}} \vdash \exists s\, \varphi_{\text{LHS}} \Rightarrow \varphi_{\text{RHS}}$		1/0/0/0/0

Fig. 3. Side-by-side proofs for the property of SUM using the Hoare logic proof system (left) and, respectively, the matching logic reachability proof system (right). The Adtl. Steps for the Hoare proof mean: Consequence rules / substitution steps. The Steps for the matching logic reachability proof mean: Transitivity / Case Analysis / Consequence / Abstraction / Circularity.

Let $\mathcal{S}_{\text{IMP}}$ be the operational semantics of IMP in Fig. 1. Now we show how the proof system in Fig. 2, using $\mathcal{S}_{\text{IMP}}$ as axioms, can derive $\mathcal{S}_{\text{IMP}} \vdash \exists s\, \varphi_{\text{LHS}} \Rightarrow \varphi_{\text{RHS}}$, the reachability rule specifying the behavior of `SUM` already discussed in Section 3, namely

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

This rule follows by Transitivity with rules 1 and 10. By Axiom $\mathbf{asgn}_{\texttt{s}}$ (Fig. 1) 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 $\varphi_{\text{LHS}} \Rightarrow \langle \texttt{LOOP},\ (\texttt{s} \mapsto 0,\ \texttt{n} \mapsto n)\rangle \wedge n \geq_{Int} 0$. Then rule 1 follows by Consequence and Abstraction with $X = \{s\}$. Rule 10 follows by Abstraction with $\{n'\}$ from rule 9. We derive rule 9 by Circularity with rules 2 and 8. Rule 2 follows by ASLF with **while**. Rule 8 follows by Case Analysis with $\varphi_{\text{true}} \Rightarrow \varphi_{\text{RHS}}$ and $\varphi_{\text{false}} \Rightarrow \varphi_{\text{RHS}}$. The latter follows by ASLF ($\mathbf{lookup}_{\texttt{n}}$, $\mathbf{op}_{>}$, $\mathbf{cond}_2$) together with some Transitivity and Consequence steps (the rule added by Circularity not needed yet). The former follows by repeated Transitivity with rules 3, 4, 5, 6. Similarly as before, rules 3, 4, 5 follow by ASLF ($\mathbf{lookup}_{\texttt{n}}$, $\mathbf{op}_{>}$, $\mathbf{cond}_1$, $\mathbf{lookup}_{\texttt{n}}$, $\mathbf{lookup}_{\texttt{s}}$, $\mathbf{op}_{+}$, $\mathbf{asgn}_{\texttt{s}}$, **seq**, $\mathbf{lookup}_{\texttt{n}}$, $\mathbf{op}_{-}$, $\mathbf{asgn}_{\texttt{n}}$, **seq**) together with Transitivity and Consequence steps. Rule 6, namely $\mathcal{S}_{\text{IMP}} \cup \{\varphi_{\text{inv}} \Rightarrow \varphi_{\text{RHS}}\} \vdash \varphi_{\text{body}} \Rightarrow \varphi_{\text{RHS}}$, follows by Axiom ($\varphi_{\text{inv}} \Rightarrow \varphi_{\text{RHS}}$) and Substitution ($\theta(n') = n' -_{Int} 1$). Note that rule 6 is in fact rule 9 with $n' -_{Int} 1$ instead of n', but now the axioms include rule 9, so we are done. Welcome to the magic of Circularity!

The table in Fig. 3 shows the number of Hoare logic language-dependent proof rules (6) and the number of Hoare logic language-independent proof rules (4 **HL-csq** rules and 51 low-level steps due to traversing the FOL formulae as part of the application of substitutions in **HL-asgn**) used in proving the property of `SUM`, for a total of 61 steps. We count the number of low-level substitution steps for the Hoare proof because those steps, which in practice do not come for free anyway, in fact do not exist in the matching logic reachability proof, being replaced by actual reasoning steps using the proof system. Fig. 3 also shows the number of matching logic reachability proof rules (80) used in proving the same example. At a first glance, the matching logic reachability proof above may appear low-level when compared to the Hoare logic proof. However, it is quite mechanical, the only interesting part being to provide the invariant pattern φ_{inv}, same like in the Hoare logic proof. Out of the 80 steps, 19 uses of the ASLF sequence (rule 6 only uses the Axiom and Substitution rules; each other ASLF step means 3 proof rule applications) and 16 of Transitivity account for most of them (72). Notice that the applications of ASLF and Transitivity are entirely syntax driven, and thus completely mechanical. There are 1 step of Case Analysis (for splitting on the symbolic condition of an `if` statement), and 2 steps of Abstraction (for eliminating existentially quantified variables), which are also mechanical. That leaves us with 4 steps of Consequence, and one step of Circularity (for dealing with the loop), which is similar to the number of steps used by the Hoare logic proof. In general, a matching logic reachability proof follows the following pattern: apply the operational rules whenever they match, except for circularities, which are given priority; when the redex is an `if`, do Case Analysis; if there are existentially quantified variables, skolemize. Our current MatchC implementation can prove the `SUM`

example automatically, as well as much more complex programs [14, 16]. Although the paper Hoare logic proofs for simple languages like IMP may look more compact, as discussed above they make (sometimes unrealistic) assumptions which need to be addressed in implementations. Finally, note that the matching logic reachability rules are more expressive than the Hoare triples, since they can specify reachable configurations which are not necessarily final. E.g.,

$$\langle \texttt{SUM},\ (\texttt{s} \mapsto s,\ \texttt{n} \mapsto n)\rangle \wedge n >_{Int} 0 \Rightarrow \langle \texttt{LOOP},\ (\texttt{s} \mapsto n,\ \texttt{n} \mapsto n -_{Int} 1)\rangle$$

is also derivable and states that if the value n of $\texttt{n}$ is strictly positive, then the loop is taken once and, when the loop is reached again, $\texttt{s}$ is n and $\texttt{n}$ is $n -_{Int} 1$.

5 From Hoare Logic Proofs to Matching Logic Reachability Proofs

Here we show how proof derivations using the IMP-specific Hoare logic proof system in Fig. 1 can be translated into proof derivations using the language-independent matching logic reachability proof system in Fig. 2 with IMP's operational semantics in Fig. 1 as axioms. The sizes of the two proof derivations are within a linear factor. We refer the reader to [15] for the proofs of the lemmas and theorems in this section.

5.1 Translating Hoare Triples into Reachability Rules

Without restricting the generality, we make the following simplifying assumptions about the Hoare triples $\{\psi\}\ \texttt{code}\ \{\psi'\}$ that appear in the Hoare logic proof derivation that we translate into a matching logic reachability proof: (1) the variables appearing in $\texttt{code}$ belong to an arbitrary but fixed finite set $\texttt{X} \subset \mathit{PVar}$; (2) the additional variables appearing in ψ and ψ' but not in $\texttt{code}$ belong to an arbitrary but fixed finite set $\texttt{Y} \subset \mathit{PVar}$ such that $\texttt{X} \cap \texttt{Y} = \emptyset$. In other words, we fix the finite disjoint sets $\texttt{X}, \texttt{Y} \subset \mathit{PVar}$, and they have the properties above for all Hoare triples that we consider in this section. Note that we used a `typewriter` font to write these sets, which is consistent with our notation for variables in PVar. We need these disjointness restrictions because, as discussed in Section 4, Hoare logic makes no theoretical distinction between program and mathematical variables, while matching logic does. These restrictions do not limit the capability of Hoare logic, since we can always pick $\texttt{X}$ to be the union of all the variables appearing in the program about which we want to reason and $\texttt{Y}$ to be the union of all the remaining variables occurring in all the state specifications in any triple anywhere in the Hoare logic proof, making sure that the names of the variables used for stating mathematical properties of the state are always chosen different from those of the variables used in programs.

Definition 6. *Given a Hoare triple* $\{\psi\}\ \texttt{code}\ \{\psi'\}$*, we define*

$$H2M(\{\psi\}\ \texttt{code}\ \{\psi'\}) \stackrel{def}{=} \exists X\ (\langle \texttt{code},\ \sigma_X\rangle \wedge \psi_{X,Y}) \Rightarrow \exists X\ (\langle \texttt{skip},\ \sigma_X\rangle \wedge \psi'_{X,Y})$$

where:

1. *$X, Y \subset \mathit{Var}$ (written using italic font) are finite sets of variables corresponding to the sets $\texttt{X}, \texttt{Y} \subset \mathit{PVar}$ fixed above, one variable x or y in Var (written using italic font) for each variable $\texttt{x}$ or $\texttt{y}$ in PVar (written using typewriter font);*

2. σ_X *is the state mapping each* $\mathtt{x} \in \mathtt{X}$ *to its corresponding* $x \in X$*; and*
3. $\psi_{X,Y}$ *and* $\psi'_{X,Y}$ *are* ψ *and respectively* ψ' *with* $\mathtt{x} \in \mathtt{X}$ *or* $\mathtt{y} \in \mathtt{Y}$ *replaced by its corresponding* $x \in X$ *or* $y \in Y$*, respectively, and each expression construct* `op` *replaced by its mathematical correspondent* op_{Int}*.*

The *H2M* mapping in Definition 6 is quite simple and mechanical, and can be implemented by a linear traversal of the Hoare triple. In fact, we have implemented it as part of the MatchC program verifier, to allow users to write program specifications in a Hoare style when possible (see, e.g., the `simple` folder of examples on the online MatchC interface at `http://fsl.cs.uiuc.edu/index.php/Special:MatchCOnline`).

It is important to note that, like $\mathtt{X}, \mathtt{Y} \subset PVar$, the sets of variables $X, Y \subset Var$ in Definition 6 are also fixed and thus the same for all Hoare triples considered in this section. For example, suppose that $\mathtt{X} = \{\mathtt{s}, \mathtt{n}\}$ and $\mathtt{Y} = \{\mathtt{oldn}, \mathtt{z}\}$. Then the Hoare triple

$$\{\mathtt{n} = \mathtt{oldn} \wedge \mathtt{n} \geq 0\}\, \mathtt{SUM}\, \{\mathtt{s} = \mathtt{oldn*(oldn+1)/2} \wedge \mathtt{n} = 0\}$$

from Section 4 is translated into the following reachability rule:

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

Not surprisingly, we can use the proof system in Fig. 2 to prove this rule equivalent to the one for `SUM` in Section 4. Using FOL and Consequence the above is equivalent to

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

which, by Substitution ($n \leftrightarrow oldn$), is equivalent to the rule in Section 4.

We also show an (artificial) example where the original Hoare triple contains a quantifier. Consider the same $\mathtt{X} = \{\mathtt{s}, \mathtt{n}\}$ and $\mathtt{Y} = \{\mathtt{oldn}, \mathtt{z}\}$ as above. Then

$$H2M(\{true\}\, \mathtt{n:=4*n+3}\, \{\exists \mathtt{z}\, (\mathtt{n} = \mathtt{2*z+1})\})$$

is the reachability rule

$$\begin{array}{l} \quad \exists s, n\, (\langle \mathtt{n:=4*n+3},\ (\mathtt{s} \mapsto s, \mathtt{n} \mapsto n)\rangle \wedge true) \\ \Rightarrow \exists s, n\, (\langle \mathtt{skip},\ (\mathtt{s} \mapsto s, \mathtt{n} \mapsto n)\rangle \wedge \exists z\, (n = 2 *_{Int} z +_{Int} 1)) \end{array}$$

Using FOL reasoning and Consequence, this rule is equivalent to

$$\exists s, n\, \langle \mathtt{n:=4*n+3},\ (\mathtt{s} \mapsto s, \mathtt{n} \mapsto n)\rangle \Rightarrow \exists s, z\, \langle \mathtt{skip},\ (\mathtt{s} \mapsto s, \mathtt{n} \mapsto 2 *_{Int} z +_{Int} 1)\rangle$$

5.2 Helping Lemmas

The following holds for matching logic in general:

Lemma 1. *If* $\mathcal{S} \vdash \varphi \Rightarrow \varphi'$ *is derivable then* $\mathcal{S} \vdash \exists X\, \varphi \Rightarrow \exists X\, \varphi'$ *is also derivable.*

The following lemma states that symbolic evaluation of IMP expressions is actually formally derivable using the matching logic reachability proof system:

Lemma 2. *If* $\mathtt{e} \in$ *Exp is an expression,* $C \in$ *Context an appropriate context, and* $\sigma \in$ *State a state term binding each program variable in PVar of* $\mathtt{e}$ *to a term of sort Int (possibly containing variables in Var), then the following sequent is derivable:*

$$\mathcal{S}_{\text{IMP}} \vdash \langle C, \sigma \rangle[\mathtt{e}] \Rightarrow \langle C, \sigma \rangle[\sigma(\mathtt{e})]$$

where $\sigma(\mathtt{e})$ *replaces each* $\mathtt{x} \in$ *PVar in* $\mathtt{e}$ *by* $\sigma(\mathtt{x})$ *(i.e., a term of sort Int) and each operation symbol* $\mathtt{op}$ *by its mathematical correspondent in the Int domain,* op_{Int}.

Intuitively, the following lemma states that if we append some extra statement to the code of φ, then the execution of the original code is still possible, making abstraction of the appended statement. This holds because of the specific (simplistic) nature of IMP and may not hold in more complex languages (for example in ones with support for reflection or self-generation of code). A direct consequence is that we can (symbolically) execute a compound statement $\mathtt{s}_1;\mathtt{s}_2$ by first executing $\mathtt{s}_1$ until we reach `skip` and then continuing from there with $\mathtt{s}_2$.

Lemma 3. *If* $\mathcal{S}_{\text{IMP}} \vdash \varphi \Rightarrow \varphi'$ *is derivable and* $\mathtt{s} \in$ *Stmt then* $\mathcal{S}_{\text{IMP}} \vdash \text{APPEND}(\varphi, \mathtt{s}) \Rightarrow \text{APPEND}(\varphi', \mathtt{s})$ *is also derivable, where* $\text{APPEND}(\varphi, \mathtt{s})$ *is the pattern obtained from* φ *by replacing each basic pattern* $\langle \mathtt{code}, \sigma \rangle$ *with the basic pattern* $\langle (\mathtt{code}; \mathtt{s}), \sigma \rangle$.

5.3 The Main Result

Theorem 2 below states that, for the IMP language, any Hoare logic proof derivation of a Hoare triple $\{\psi\}\ \mathtt{code}\ \{\psi'\}$ yields a matching logic reachability proof derivation of the corresponding reachability rule $H2M(\{\psi\}\ \mathtt{code}\ \{\psi'\})$. This proof correspondence is constructive and the resulting proof derivation is linear in the size of the original proof derivation. For example, to generate the matching logic reachability proof corresponding to a proof step using the Hoare logic proof rule for `while` loop, **HL-while**, we do the following (see the proof of Theorem 2 in [15] for all the details):

1. We inductively assume a proof for the reachability rule corresponding to the Hoare triple for the `while` loop body;
2. We apply the Axiom step with **while** (Fig. 1), followed by Substitution, Logic Framing, and Lemma 1, and this way we "unroll" the `while` loop into its corresponding conditional statement (in the logical context set by the Hoare triple);
3. Since the conditional statement contains the original `while` loop in its true branch and since 2. above does not use Reflexivity, we issue a Circularity proof obligation and thus add the claimed reachability rule for `while` to the set of axioms;
4. We "evaluate" symbolically the condition, by virtue of Lemma 2;
5. We apply a Case Analysis for the conditional, splitting the proof task in two subtasks, the one corresponding to the false condition being trivial to discharge;
6. To discharge the care corresponding to the true condition, we use the proof given by 1. by virtue of Lemma 3, then the Axiom for **seq**, and then the reachability rule added by Circularity and we are done.

Theorem 2. *(see [15] for the proof) Let $\mathcal{S}_{\text{IMP}}$ be the operational semantics of IMP in Fig. 1 viewed as a matching logic reachability system, and let $\{\psi\}\,\texttt{code}\,\{\psi'\}$ be a triple derivable with the IMP-specific Hoare logic proof system in Fig 1. Then we have that $\mathcal{S}_{\text{IMP}} \vdash H2M(\{\psi\}\,\texttt{code}\,\{\psi'\})$ is derivable with the language-independent matching logic proof system in Fig. 2.*

Theorem 2 thus tells us that anything that can be proved using Hoare logic can also be proved using the matching logic reachability proof system. Furthermore, it gives us a novel way to prove soundness of Hoare logic proof systems, where the low-level details of the transition system corresponding to the target programming language, including induction on path length, are totally avoided and replaced by an abstract, small and fixed proof system, which is sound for all languages.

5.4 Adding Recursion

In this section we add procedures to IMP, which can be mutually recursive, and show that proof derivations done with the corresponding Hoare logic proof rule can also be done using the generic matching logic proof system, with the straightforward operational semantics rule as an axiom. We consider the following syntax for procedures:

$$\begin{array}{l} \mathit{ProcedureName} \;::=\; \texttt{proc} \mid \ldots \\ \mathit{Procedure} \;::=\; \mathit{ProcedureName}()\;\mathit{Stmt} \\ \mathit{Stmt} \;::=\; \ldots \mid \mathit{ProcedureName}() \end{array}$$

Our procedures therefore have the syntax "$\texttt{proc}()\,\texttt{body}$", where $\texttt{proc}$ is the name of the procedure and $\texttt{body}$ the body statement. Procedure invocations are statements of the form "$\texttt{proc}()$". For simplicity, and to capture the essence of the relationship between recursion and the Circularity rule of matching logic, we assume only no-argument procedures.

The operational semantics of procedure calls is trivial:

$$\textbf{call} \quad \texttt{proc}() \;\Rightarrow\; \texttt{body} \qquad \text{where "}\texttt{proc}()\,\texttt{body}\text{" is a procedure}$$

The Hoare logic proof rule needs to take into account that procedures may be recursive:

$$\frac{\mathcal{H} \cup \{\psi\}\,\texttt{proc}()\,\{\psi'\} \vdash \{\psi\}\,\texttt{body}\,\{\psi'\}}{\mathcal{H} \vdash \{\psi\}\,\texttt{proc}()\,\{\psi'\}} \quad \text{where "}\texttt{proc}()\;\texttt{body}\text{" is a procedure}$$

This rule states that if the body of a procedure is proved to satisfy its contract while assuming that the procedure itself satisfies it, then the procedure's contract is indeed valid. If one has more mutually recursive procedures, then one needs to apply this rule several times until all procedure contracts are added to the hypothesis $\mathcal{H}$, and then each procedure body proved. The rule above needs to be added to the Hoare logic proof system in Fig. 1, but in order for that to make sense we need to first replace each Hoare triple $\{\psi\}\,\texttt{code}\,\{\psi'\}$ in Fig. 1 by a sequent "$\mathcal{H} \vdash \{\psi\}\,\texttt{code}\,\{\psi'\}$".

Theorem 3. *(see [15] for the proof) Let $\mathcal{S}_{\text{IMP}}$ be the operational semantics of IMP in Fig. 1 extended with the rule **call** for procedure calls above, and let $\mathcal{H} \vdash \{\psi\}\,\texttt{code}\,\{\psi'\}$ be a sequent derivable with the extended Hoare logic proof system. Then we have that $\mathcal{S}_{\text{IMP}} \cup H2M(\mathcal{H}) \vdash H2M(\{\psi\}\,\texttt{code}\,\{\psi'\})$ is derivable with the matching logic reachability proof system in Fig. 2.*

6 Conclusion

Matching logic reachability provides a sound and language-independent program reasoning method, based solely on the operational semantics of the target programming language [16]. This paper addressed the other important aspect of matching logic reachability deduction, namely its completeness. A mechanical and linear translation of Hoare logic proof trees into equivalent matching logic reachability proof trees was presented. The method was described and proved correct for a simple imperative language with both iterative and recursive constructs, but the underlying principles of the translation are general and should apply to any language. The results presented in this paper have two theoretical consequences. First, they establish the relative completeness of matching logic reachability for a standard language, by reduction to the relative completeness of Hoare logic, and thus show that matching logic reachability is at least as powerful as Hoare logic. Second, they give an alternative approach to proving soundness of Hoare logics, by reduction to the generic soundness of matching logic reachability.

Acknowledgements: We thank Michael Whalen and Cesare Tinelli for the interesting discussions we had at Midwest Verification Day 2011, which stimulated this research. We also thank the members of the $\mathbb{K}$ team (`http://k-framework.org`) and the anonymous reviewers for their valuable comments on a previous version 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 (2011)
2. Berry, G., Boudol, G.: The chemical abstract machine. Theoretical Computer Science 96(1), 217–248 (1992)
3. Blazy, S., Leroy, X.: Mechanized semantics for the Clight subset of the C language. Journal of Automated Reasoning 43(3), 263–288 (2009)
4. Clavel, M., Durán, F., Eker, S., Meseguer, J., Lincoln, P., Martí-Oliet, N., Talcott, C.: All About Maude, LNCS, vol. 4350. Springer (2007)
5. Cook, S.A.: Soundness and completeness of an axiom system for program verification. SIAM Journal on Computing 7(1), 70–90 (1978)
6. Ellison, C., Roşu, G.: An executable formal semantics of C with applications. In: POPL. pp. 533–544 (2012)
7. Felleisen, M., Findler, R.B., Flatt, M.: Semantics Engineering with PLT Redex. MIT (2009)
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. Jacobs, B.: Weakest pre-condition reasoning for Java programs with JML annotations. The Journal of Logic and Algebraic Programming 58(1-2), 61–88 (2004)
10. Liu, H., Moore, J.S.: Java program verification via a JVM deep embedding in ACL2. In: TPHOLs. LNCS, vol. 3223, pp. 184–200 (2004)
11. Mosses, P.D.: CASL Reference Manual, LNCS, vol. 2960. Springer (2004)
12. Nipkow, T.: Winskel is (almost) right: Towards a mechanized semantics textbook. Formal Aspects of Computing 10, 171–186 (1998)
13. Roşu, G., Şerbănuţă, T.F.: An overview of the K semantic framework. The Journal of Logic and Algebraic Programming 79(6), 397–434 (2010)

14. Roşu, G., Ştefănescu, A.: Matching logic: A new program verification approach (NIER track). In: ICSE. pp. 868–871 (2011)
15. Roşu, G., Ştefănescu, A.: From Hoare logic to matching logic reachability. Tech. Rep. http://hdl.handle.net/2142/31335, Univ. of Illinois (June 2012)
16. Roşu, G., Ştefănescu, A.: Towards a unified theory of operational and axiomatic semantics. In: ICALP. LNCS, Springer (2012), to appear.
17. Roşu, G., Ellison, C., Schulte, W.: Matching logic: An alternative to Hoare/Floyd logic. In: AMAST. LNCS, vol. 6486, pp. 142–162 (2010)
18. Sasse, R., Meseguer, J.: Java+ITP: A verification tool based on Hoare logic and algebraic semantics. Electronic Notes in Theoretical Computer Science 176(4), 29–46 (2007)